JACO HOFMANN

AN IMPROVED FRAMEWORK FOR AND CASE STUDIES IN FPGA-BASED APPLICATION ACCELERATION – COMPUTER VISION, IN-NETWORK PROCESSING AND SPIKING NEURAL NETWORKS

AN IMPROVED FRAMEWORK FOR AND CASE STUDIES IN FPGA-BASED APPLICATION ACCELERATION – COMPUTER VISION, IN-NETWORK PROCESSING AND SPIKING NEURAL NETWORKS

submitted in fulfilment of the requirements for the degree of Doctor of Engineering (Doktoringenieur, Dr.-Ing.)

Doctoral thesis
by Jaco Hofmann from Hamburg

Disputation: 18.12.2019
Department of Computer Science
TU Darmstadt

First assessor: Prof. Dr.-Ing. Andreas Koch
Second assessor: Prof. Dr.-Ing. Mladen Berekovic
Darmstadt – D 17

Jaco Hofmann: *An Improved Framework for and Case Studies in FPGA-Based Application Acceleration – Computer Vision, In-Network Processing and Spiking Neural Networks*.
Dissertation. Technische Universität Darmstadt. 2019.

Please cite this document as:
URN: urn:nbn:de:tuda-tuprints-103551
URL: https://tuprints.ulb.tu-darmstadt.de/id/eprint/10355

This document is provided by tuprints, the e-publishing-service of TU Darmstadt.
https://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

ABSTRACT

Field Programmable Gate Arrays (FPGAs) are a new addition to the world of data center acceleration. While the underlying technology has been around for decades, their application in data centers slowly starts gaining traction. However, there are myriad problems that hinder the widespread application of FPGAs in the data center. The closed source tool chains result in vendor lock-in and unstable tool flows. The languages used to program FPGAs require different design processes which are not easily learned by software developers. Compared to commodity solutions using CPUs and GPUs, FPGAs are more expensive and more time consuming to develop for. All of this and more make FPGAs a tough sell to people in need of task acceleration.

Nonetheless, FPGAs also offer an opportunity to develop faster accelerators with a smaller energy envelop for rapidly changing applications. This work presents a solution to FPGA abstraction using the TaPaSCo framework. TaPaSCo simplifies moving between different FPGA architectures and automates scaling of accelerators across a multitude of devices. In addition, the framework provides a homogenized way of interacting with the accelerators.

This thesis presents applications where FPGAs offer many benefits in the data center. Applications such as Semi-Global Block Matching which are difficult to compute on CPUs and GPUs due to the specific data transfer patterns, can be implemented highly efficiently an FPGAs. The presented work achieves over 35× of speedup on FPGAs compared to implementations of GPUs.

FPGAs can also be used to improve network efficiency in the data center by replacing central network components with smart switches. The work presented here achieves up to 7× speedup over a classical distributed software implementation in a hash join scenario.

Furthermore, FPGA can be used to bring new storage technologies into the data center by providing highly efficient consensus services right inside the network. The presented work shows that fetching pages remotely using a FPGA accelerated consensus system can be done as fast as 10 µs over the network which is only 55 % of a conventional solution. These results make non-volatile network storage solutions as replacement for main memory viable.

Lastly, this thesis presents a way of simulating parts of a brain with a very high level accuracy using FPGA. The spiking neural networks employed in the accelerator can benefit the research of brain functionality. The accelerator is capable of handling tens of thousands of neurons with a strict real time requirement of 50 µs per simulation step.

ZUSAMMENFASSUNG

Field Programmable Gate Arrays (FPGAs) sind eine neue Art von Beschleunigern in der Welt der Rechenzentren. Während die grundlegende Technologie bereits seit Jahrzehnten verwendet wird, ist ihr Einsatz in Rechenzentren neu. Sie setzen sich dort aufgrund einer Vielzahl von Problemen nur langsam durch. Die proprietäre Software, die die Nutzung von FPGAs ermöglicht, sorgt dafür, dass zum einen der Wechsel des Herstellers schwierig ist und zum anderen die Tools oft von Stabilitätsproblemen geplagt werden. Die Programmiersprachen für FPGAs benötigen eine völlig andere Entwurfsmethodik, was deren Verwendung für Softwareentwickler erschwert. Im Vergleich zu verbreiteten Lösungen, die auf CPUs und GPUs basieren, ist die Entwicklungszeit für FPGA-basierte Lösungen höher. All diese Aspekte verzögern den Durchbruch von FPGAs in Rechenzentren.

Trotzdem bieten FPGAs die Möglichkeit, effizientere und schnellere Beschleuniger für eine große Zahl von sich schnell ändernden Applikationen zu entwickeln. Diese Dissertation präsentiert TaPaSCo als eine Lösung zur Abstraktion von FPGAs. Das Framework erleichtert das Wechseln zwischen verschiedenen FPGA-Architekturen. Außerdem automatisiert die Software die Skalierung der Beschleuniger für eine große Anzahl von Plattformen. Zusätzlich ermöglicht TaPaSCo eine einheitliche Interaktion mit verschiedenen FPGAs.

Des Weiteren präsentiert diese Arbeit Anwendungen, in denen FPGAs einen deutlichen Mehrwert in Rechenzentren bieten können. Anwendungen wie das Semi-Global Block Matching sind durch ihre Kommunikationsmuster auf CPUs und GPUs schwer zu berechnen. Auf FPGAs können diese Muster allerdings sehr effizient implementiert werden. Der vorgestellte Beschleuniger erreicht eine Verbesserung von über 35× im Vergleich zu einer Implementierung auf GPUs.

Andere Einsatzgebiete für FPGA finden sich in den Netzwerken der Rechenzentren, in denen sie als intelligente Switches die Effizienz der Datenkommunikation steigern können. Bei einem verteilten Hash Join, der eine typische Arbeitslast von verteilten Datenbanken ist, erreicht eine Implementierung in einem FPGA eine Verbesserung von 7× über eine klassische Implementierung.

Zudem können FPGAs eingesetzt werden, um völlig neue Speichertechnologien in die Datenzentren zu bringen, indem sie an zentraler Stelle Konsensus-Dienstleistungen erbringen. Die vorgestellte Arbeit zeigt, dass eine Speicherseite, die über das FPGA gesteuerte Konsensus-Netzwerk abgefragt wird, eine Latzen von nur 10 µs hat, was einer Verschlechterung gegenüber einer lokalen Abfrage von nur 55 % entspricht. Solch ein Konsensus-System ermöglicht es, nicht-

flüchtige Netzwerkspeicher als Ersatz für lokale Hauptspeicher zu verwenden.

Zum Abschluss präsentiert diese Dissertation einen Beschleuniger, der auf FPGAs die Neuronen eines Gehirns sehr genau simulieren kann. Die “Spiking Neural Networks”, die dafür verwendet werden, können in der Gehirnforschung als Ersatz zur in-vivo Forschung eingesetzt werden. Der Beschleuniger kann mehrere zehntausend Neuronen pro Simulationsschritt berechnen, während er eine starke Echtzeitbedingung von 50 µs einhält.

PUBLICATIONS

CONFERENCE

[1] Jaco Hofmann, Jens Korinth, and Andreas Koch. "A Scalable High-Performance Hardware Architecture for Real-Time Stereo Vision by Semi-Global Matching." In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. Best Paper Runner-Up. 2016.

[2] Jaco Hofmann, Jens Korinth, and Andreas Koch. "A Scalable Latency-Insensitive Architecture for FPGA-Accelerated Semi-Global Matching in Stereo Vision Applications." In: *IEEE Proc. International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2016.

[3] Jaco Hofmann, A. Zjajo, and R. van Leuken. "Multi-chip dataflow architecture for massive scale biophysically accurate neuron simulation." In: *38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. 2016.

[4] Lukas Sommer, Julian Oppermann, Jaco Hofmann, and Andreas Koch. "Synthesis of Interleaved Multithreaded Accelerators from OpenMP Loops." In: *2017 International Conference on Reconfigurable Computing and FPGAs (ReConFig'17)*. 2017.

[5] Huynh Tu Dang, Jaco Hofmann, Yang Liu, Marjan Radi, Dejan Vucinic, Robert Soulé, and Fernando Pedone. "Consensus for Non-Volatile Main Memory." In: *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. IEEE. 2018, pp. 406–411.

[6] Carsten Heinz, Yannick Lavan, Jaco Hofmann, and Andreas Koch. "A Catalog and In-Hardware Evaluation of Open-Source Drop-In Compatible RISC-V Softcore Processors." In: *IEEE Proc. International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE. 2019.

[7] Jaco Hofmann, Lasse Thostrup, Tobias Ziegler, Carsten Binnig, and Andreas Koch. "High-Performance In-Network Data Processing." In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2019, Los Angeles, United States.* 2019.

[8] Jens Korinth, Jaco Hofmann, Carsten Heinz, and Andreas Koch. "The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems." In: *International Symposium on Applied Reconfigurable Computing (ARC)*. 2019.

[9] Johannes Krude, Jaco Hofmann, Matthias Eichholz, Klaus Wehrle, Andreas Koch, and Mira Mezini. "Online Reprogrammable Multi Tenant Switches." In: *1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms (CoNEXT ENCP'19)*. ACM. 2019.

[10] Micha Ober, Jaco Hofmann, Lukas Sommer, Lukas Weber, and Andreas Koch. "High-Throughput Multi-Threaded Sum-Product Network Inference in the Reconfigurable Cloud." In: *Fifth International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 2019.

JOURNAL

[11] Amir Zjajo, Jaco Hofmann, Gerrit Jan Christiaanse, Martijn Van Eijk, Georgios Smaragdos, Christos Strydis, Alexander de Graaf, Carlo Galuzzi, and Rene van Leuken. "A real-time reconfigurable multichip architecture for large-scale biophysically accurate neuron simulation." In: *IEEE transactions on biomedical circuits and systems* 12.2 (2018), pp. 326–337.

BOOKCHAPTER

[12] Jaco Hofmann. "Real-Time Multi-Chip Neural Network for Cognitive Systems." In: ed. by A. Zjajo and R. van Leuken. Real-Time Multi-Chip Neural Network for Cognitive Systems. River Publishers, 2019. Chap. MultiChip Dataflow Architecture for Massive Scale Biophysically Accurate Neuron Simulation.

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor, Prof. Dr.-Ing. Andreas Koch, for his unwavering support. He provided me with a great environment where I could follow my research goals, and he was always ready to give advice whenever needed. Additionally, he always had a new interesting project to tackle.

I would like to express my gratitude to Prof. Dr.-Ing. Mladen Berekovic for taking the time out of his busy schedule and being the second assessor of this work.

Special thanks are reserved for my current and former colleagues with whom I have worked through many interesting challenges.

I am thankful for my friends for being there when I needed them. They made me see things differently and helped me through some tough times. Life would not be the same without them.

I am indebted to my family for their adamant backing. Without them I would not have made it this far.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

ANN Artifical Neural Network

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

CLB Configurable Logic Block

DMA Direct Memory Access

DSE Design Space Exploration

DSL Domain Specific Language

DSP Digital Signal Processor

FF Flip-Flop

FPGA Field Programmable Gate Array

GPGPU General-Purpose Computing on Graphics Processing Units

HCL Hardware Construction Language

HDL Hardware Description Language

HLS High-level Synthesis

INP In-Network Processing

IP Intellectual Property

LUT Lookup Table

NOC Network on Chip

PCIE Peripheral Component Interconnect Express

PE Processing Element

RTL Register-Transfer Level

SGBM Semi-Global Block Matching

SNN Spiking Neural Network

SOC System-on-Chip

TPC Threadpool Composer

Part I

FIELD PROGRAMMABLE GATE ARRAYS

1 INTRODUCTION

Today data centers are dominated by two types of processing hardware: Firstly there are general purpose CPUs that are used for a wide area of applications and are the driving force behind a lot of processing done today. They power the Internet by running web-servers and other required infrastructure such as databases. They are used to simulate physical processes, explore the universe or predict the weather. CPUs might not be the fastest when looking at specific applications, however, they can reliably process pretty much any workload thrown at them. For a long time (and even in some new data centers built today) CPUs have been the only processors available in data centers. The ever increasing demand for performance lead to the advent of General-Purpose Computing on Graphics Processing Units (GPGPU)[34]. Whereas CPUs currently process up 128 threads[4] at the same time with very little parallelism (excluding techniques like SIMD or SMT), GPUs can process a huge number of threads simultaneously and can schedule new threads at an instant to hide memory latency. The new contender in the processing game has rapidly grown in popularity especially through their efficient integration into popular machine learning frameworks.

Common to both types of accelerators is their easy programmability, straight forward upgrade paths and very good tool and community support. They are usually programmed in some kind of high-level programming language, or in the case of GPUs, abstracted away through powerful frameworks for certain tasks. Parallelism can be automatically exploited through techniques such as OpenMP[25] and OpenCL[117]. Programs written for a certain CPU architecture, commonly x86, can be run on CPUs introduced much later than the codes inception. Similarly, kernels written for a given GPU generation can be run on newer generations or different models of the same generation with reasonably good performance. All the necessary tools to access both architectures are widely available and in the majority of cases even as open-source software.

The new contender in the field of datacenter application acceleration are FPGA. They are certainly not a new invention; the earliest models had been available as early as 1983 [128]. However, their use was limited to certain applications and they never entered the mainstream of processing due to a variety of reasons. FPGA always excelled at tasks such as low latency processing (for example high frequency trading), glue logic tasks or high throughput streaming. Until 2014 [22] they did not compete with the general purpose processors, namely

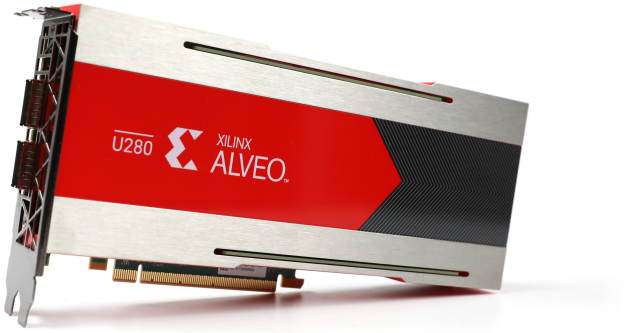


Figure 1.1: The Xilinx Alveo U280 Data Center Accelerator Card already looks the part to be used in a data center. Picture taken from the manufacturer at https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

CPUs and GPUs, in the datacenter space for application acceleration. FPGA competed more with custom Application-Specific Integrated Circuit (ASIC). If the volume required was relatively small, for example in the automotive industry, an FPGA was a good alternative before spinning silicon which leads to high initial costs. To help to understand the difference between CPUs, GPUs, ASIC and FPGA better, a short description of the latter is presented in Chapter 2.

Nowadays FPGAs push into the datacenter as well as they promise interesting features: Higher performance and lower energy consumption are just two of the many advantages over traditional compute nodes. Microsoft was among the firsts bringing FPGA into the cloud in Project Catapult in 2014 [22]. The initial results in search query processing were impressive. A 2x speedup in throughput and a 29 % reduction in latency. About half the servers used for query processing can be cut leading to energy and cost savings. Later on different cloud providers such as Amazon and Huawai brought user programmable FPGAs to their cloud offerings. In previous years experimenting with FPGA required not only in-depth hardware development knowledge but also substantial investment into FPGA development boards. Contrary to the software landscape the barrier to entry was very high. With the new cloud offerings, however, it is as easy as starting an FPGA instance on one's favorite cloud provider to get started.

And this is where a bright future of FPGA accelerated datacenters with low energy footprint and high performance could start. Sadly, there are many caveats that apply when using FPGA.

1. FPGA are not programmed in software but represent hardware. The thought process is very different and former software developers can not simply switch to the new accelerator as they could with GPUs.

2. It is not easy to scale up computation across multiple FPGAs. This is not too different from CPU or GPU based implementations, but the research is far less advanced (e.g. there is no FPGA-MPI).

3. Moving from one FPGA inside a family to another is difficult. Moving from one FPGA to a different family is even more difficult. Moving to a different FPGA vendor altogether is often not feasible due to the high associated costs.

4. Compilation, which is a relatively pain-free process in the software world, becomes a major bottleneck. The roughly equivalent process of synthesis and place&route (See Chapter 2) can take days for the FPGA used in datacenters today. Errors in the toolflow can occur at any step, and even after multiple days of runtime and for various reasons, making the run invalid.

5. There is no debugging with tools like GDB or Nsight. There is not even "printf-Debugging" available. Finding errors in ones design on the chip can be a big problem. Simulation is a major time saver but can not find all problems depending on how accurate the simulation environment is.

6. Many systems inside modern PCs, such as the PCIe bus, are rarely properly secured. Techniques such as IOMMUs are frequently disabled. A malicious design on the FPGA can read all the host memory if no other precautions have been taken.

7. Outside of the datacenter environment, FPGAs are difficult to set up. On premise hosting requires much work to get right.

This brief list gives an idea of the problems datacenter providers and developers targeting FPGA must face. The FPGA community has been battling these problems for the past decades but many of them are still open problems and require further research.

Let's take for example the problems described in Item 1. Traditionally, FPGAs have been programmed through Hardware Description Languages (HDLs). In an HDL, most commonly VHDL and Verilog, the designer describes hardware at different, but very low, levels of abstraction. At the lowest layer one might describe individual primitives of the FPGA and their connection. This would be the layer used to describe the FPGA design after Synthesis and before Place&Route. A human developer would usually, using the same language, describe an FPGA design one level higher in the register-transfer-level, which describes registers and the combinational logic between them. This level still requires a lot of knowledge about hardware and, while the languages look deceivingly like software programming languages, describes a totally different method of computation. Software developers trying to utilize the power of FPGA must completely relearn a different pattern of thinking to be productive in HDLs. Additionally,

the traditional HDL have a tendency to confuse newcomers as they provide features that are valid language constructs and simulate fine, but can not be synthesized into hardware by the tools later on.

A new generation of HDL, sometimes (mainly by the Chisel authors[10]) called Hardware Construction Language (HCL), has been developed in the past decade. The purpose of these languages is also to closely describe hardware, but in a much more convenient fashion. These languages assist the designer with tedious tasks. For example they can automatically connect up the large number of signals found in common bus-interfaces such as AXI. In general, the languages in this group do not provide a higher abstraction level compared to Verilog and VHDL. One notable exception to this rule is Bluespec which uses a technique called `guarded-atomic-actions` to improve the abstraction level and increase productivity. A short overview of common HDL and HCL is given in Chapter 3.

Even the newer generation of HCL still requires fundamental hardware knowledge. The FPGA vendors are aware of the steep learning curve and try to alleviate certain aspects by providing a group of languages and tools usually grouped as High-level Synthesis (HLS). In contrast to classical HDL the promise of HLS is that the former software developer does not have to learn about hardware but can program in familiar languages. In academia many tools such as [57] try to synthesize run-of-the-mill C code into hardware. The industry focuses on tools like Vivado HLS to fulfill the same task. Common languages used in HLS are C and C++ as classical software languages and SystemC as a library built on top of C++ incorporating certain hardware design aspects. While all these tools promise ease-of-use and a shallow learning curve, they generally cannot keep that promise. For most applications the programmer has to adopt the code to the HLS tool and convert it into some compatible form to achieve good performance. The resulting source code often looks nothing like the original and the programmer might have been better off using HLS directly[102]. A second group is built around the widely used heterogeneous platform targeted language OpenCL. However, the same caveat applies as an OpenCL program written for GPU which will run poorly on FPGA. The group of HLS is described in detail in Chapter 3.

Another approach in the HLS domain are Domain Specific Languages (DSLs). Instead of synthesizing general purpose languages, a subset specific to a certain use case is used. The use case might be machine learning, image processing or cryptography. As the scope is relatively narrow, dedicated hardware accelerators can be provided and combined according to the user requirements. This approach shows very promising results [45] and could become as popular as they have become with machine learning frameworks for GPU.

With whatever approach chosen, there is no way around synthesis and place&route. The tools used for these steps are completely vendor

locked and closed source. Only very recent developments allowed for some interaction with those tools [75], previously only reverse engineering was possible [76]. Accordingly, a user is mostly at the mercy of the FPGA manufacturer. The process then takes from 30 minutes to multiple days depending on the complexity of the design and the target FPGA. In the software world a user might notice inefficient design with slow performance and can then proceed with tracing and other techniques to incrementally improve the design. This incremental process is a lot slower considering the lengthy process of bringing the design on the FPGA. Accordingly, the user has to hope the HLS tool does things efficiently or, when using HDL, have a large knowledge about certain features of the underlying architecture and has to know exactly what construct in the HDL leads to what structure after device mapping. Inefficient designs on the FPGA will achieve poor clock frequencies or are very big, often a combination of both.

Even the most experienced designers will have to debug their design. In software there are many very well supported tools from a variety of manufacturers. These tools give in-depth information about the execution of a program and allow a look down to the single instruction level. Program debugging, even by using these tools, is often the lengthiest part in software development. However, the process is relatively annoyance free, especially in single-threaded applications. The FPGA world is very different. There are no traditional debugging tools for FPGA as the concept of, for example, single-stepping hardly translates onto the highly parallel execution of hardware. Accordingly, different techniques have to be used. Even today the most common approach is to simulate the design and either look at the individual signals at the right time, as shown in Figure 1.2, or use methods akin to `printf`-debugging, where console output is generated depending on the state of the accelerator. Certain methods, such as fuzzy testing, cropped up over the years to help notice errors.

However, the process of finding the errors location is still very frustrating. In addition to the lack of comfortable debugging tools, the simulations employed are also painstakingly slow. Depending on the degree of accuracy, the simulation of the initial phases of a DDR3 connection can take upwards of 30 minutes. If the error is not in the initialization process, the real simulation until the error occurs might also take as long. As some errors only occur when the simulation replicates the behavior of the real device accurate enough, there is sometimes no way around these simulation times. Newer languages such as Bluespec or SystemC rapidly increase simulation speed for code written in the respective languages. However, often certain Intellectual Property (IP) is only provided in Verilog or VHDL and cannot be simulated with the language specific simulators.

All in all simulation is still relatively fast compared to what happens if an error only occurs on the device but not in simulation. Those errors

can usually be reproduced in accurate simulation. If certain behavior on the device is unexpected, there is no way for the simulation to mimic that aspect. These situations are the most time-consuming parts of FPGA debugging. Bitstreams to reproduce the error have to be generated. The error has to be localized. Special IP that acts as a internal logic analyzer has to be instantiated, and the error has to be captured. After all that process the simulation has to be enhanced to cover the error, hopefully, detected from the traces. After finding and fixing the error based on the simulation, the whole process has to start all over again. All of this is just a very brief overview of what debugging means in a FPGA environment. In reality, this process typically takes much longer as certain problems for example only occur in very high-performance cases which are difficult to capture and simulate.

When it comes down to performance it does not look much better. In the software world many well-tuned tracing frameworks exist to find out where exactly performance is lost. In the FPGA domain all the required infrastructure has to be provided by the designer itself. Methods such as performance counters offer some way of telling if something is amiss. However, they are usually just very vague tools and can only provide hints but no certainty. Performance problems indicated by the performance counters then require another lengthy simulation, bitstream, repeat process to lead to results. These changes might then have unexpected consequences on the timing or resource utilization and the whole process starts all over again.

Let's assume the designer generated a design using his favorite HDL or HLS tool. Everything works alright and at a high performance. As is the case with all types of accelerators a new FPGA generation comes around. For software programmers the cases would be clear. In the worst case the program has to be recompiled to work on the new architecture at the optimal performance. All the infrastructure, however, stays the same and changes in the code are usually not necessary. Later, some changes to the code can further improve performance, but are not required. Looking at the FPGA world again, there are major differences. With a new generation of FPGA most of the supporting infrastructure has to be changed. A DDR3 controller might be interfaced completely different from a DDR4 controller, the new generation might require a different PCIe interface core and so on. These changes are usually not only limited to the infrastructure but also radiate into the accelerator itself when changes to the utilized communication schemes are made. Accordingly, going to a new generation is often a very slow and error prone process. All the changes must be tested again with the above-mentioned process. Ultimately, changing from one generation of FPGA to another one is often very expensive. Apart from the set-up related problems, there might also be new features that have to be used for optimal performance. In the software world

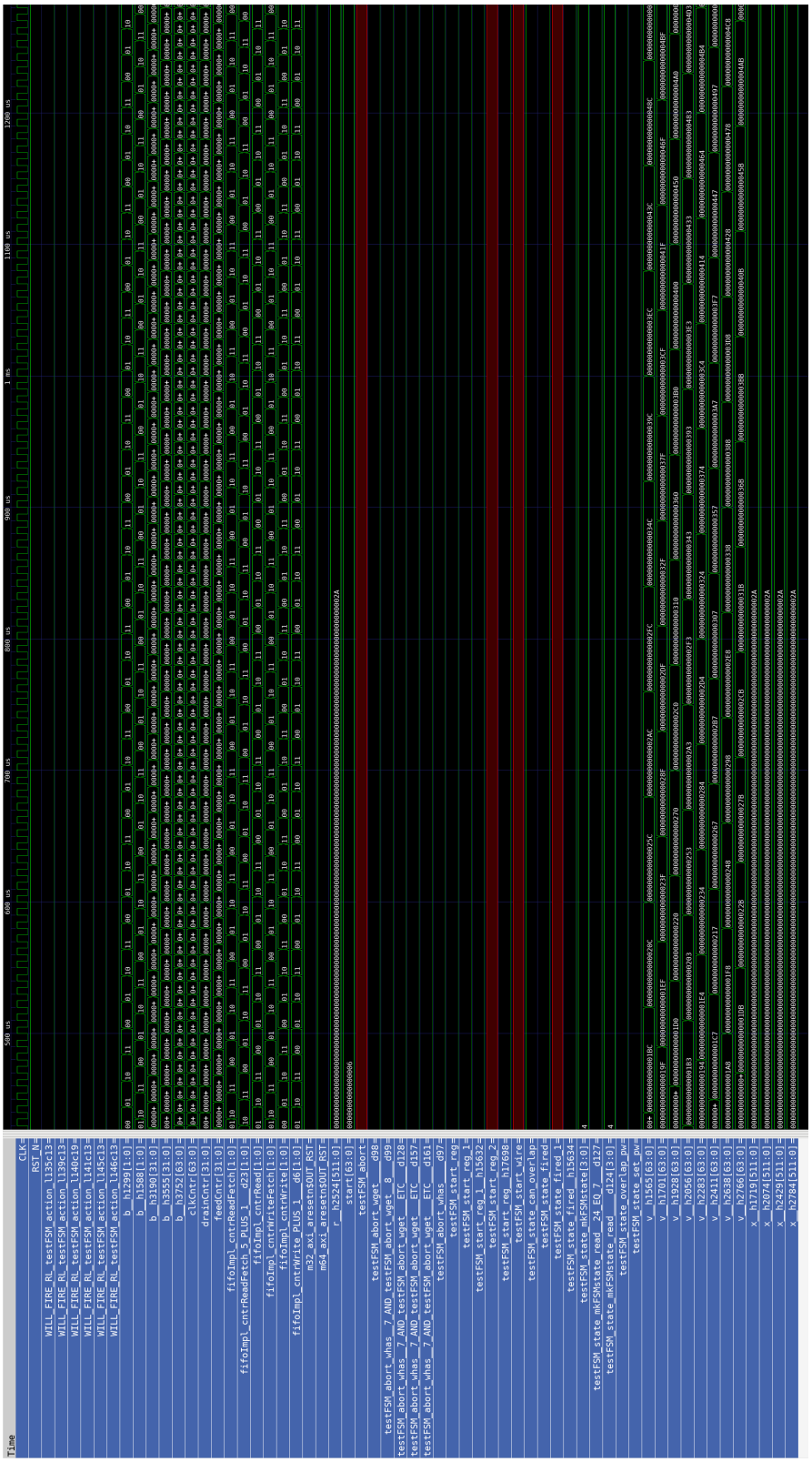


Figure 1.2: Debugging a Bluespec generated IP using signal waveforms for transfers from host to device memory as used in TaPaSCo.

these features are usually automatically employed by the compiler. For FPGAs on the other hand the novel technologies might require in-depth changes of any part of the design which triggers the aforementioned resynthesize-loop again.

The preceding paragraphs paint a rather bleak picture of FPGA as a competitor to traditional computing devices and there is certainly a long way to go, apart from availability, for mainstream appeal. However, the recent years have shown that there are many applications that are not as easily mappable on CPUs or GPUs. Even problems well suited for GPUs such as neural network inference, FPGAs offer great power savings and an increased throughput. Techniques like playing with single bit precision are easy to do on FPGAs but not possible on traditional computing devices.

This thesis provides an overview of the current accelerator development landscape for FPGA. Subsequentially, I will introduce a tool capable of abstracting away most of the changes between FPGA generations, enabling the designer to use IP across them. Lastly, I will present different application scenarios where FPGA provide added value compared to other types of computation devices.

1.1 THESIS CONTRIBUTIONS

- A hardware and software abstraction layer offered as part of the open-source TaPaSCo framework that can abstract generational and interface changes between different FPGA away from the designer.
- A very high-performance stereo vision accelerator for SGBM that delivers much higher framerates at lower latencies compared to CPU or GPU based implementation while, at the same time, being highly configurable for low power embedded as well as high performance datacenter applications.
- An FPGA in the middle of a network, acting as a switch, for implementing a low latency consensus protocol directly in the network to bring reliable next generation storage class memories to the data center.
- A Hash-Join acceleration for distributed databases directly on the FPGA acting as a switch to prototype next generation, state capable, programmable switches.
- Utilizing the flexible interconnection schemes of FPGA to accurately simulate brain neuron function in real-time using Spiking Neural Networks (SNN).

1.2 THESIS OUTLINE

The first part of this work gives an overview of today's FPGA ecosystem. In Chapter 2 FPGAs are introduced as accelerators, shortly describing their main components and how designs are mapped onto them. In Chapter 3 the two major types of design languages for FPGA are presented to give an overview over the current landscape and provide insight needed for later chapters. Lastly, in Chapter 4 certain aspects of traditional FPGA design are compared to a more modern and datacenter focused approach. In Chapter 5 TaPaSCo, our System-on-Chip generator that can help make FPGA more accessible, is introduced.

The second part of this work revolves around specific accelerator designs. Common to all the designs is that FPGAs can stand out compared to more traditional computing elements. In Chapter 6 an accelerator for SGBM is shown that can provide much higher throughput, lower latency and less energy consumption compared to CPU and GPU. In Chapter 7 FPGAs are moved into the network and two different applications for the devices are presented. In both applications the FPGA functions as the switch which enables processing of data directly in the network without the need for costly extra data movement. In Chapter 10 FPGAs are used as a tool to simulate highly parallel and biophysically accurate neurons of a simulated brain for research purposes. The FPGA shines in this task as it meets very tight latency requirements and is able to implement difficult communication schemes.

Lastly, in Chapter 11, the results of this thesis are summarized and the fundamental aspects learnt as part of this work are enumerated.

2

FPGAS AS TARGET FOR ACCELERATOR DEVELOPMENT

Chapter 1 has already given a brief overview of what it takes to target an FPGA for application acceleration. The introduction contained fundamental aspects of FPGA design such as the tedious synthesis and place&route processes. However, targeting an FPGA without the necessary understanding of the structure of FPGA is futile. Accordingly, this chapter describes important aspects of a modern FPGA necessary to understand the challenges of FPGA design. This chapter cannot give a comprehensive in-depth view of all aspects involved in FPGA designs. Please refer to the literature, for example [24, 81], for in-depth information not covered here.

2.1 BASIC BUILDING BLOCKS

The basic functionality of a FPGA is to replicate any logic function in a reconfigurable manner. So, what is a logic function? Let's take as example the following schematic in Figure 2.1. The schematic can be read from left to right. The function takes three inputs and produces one output. The gate on the left is called an AND gate and produces true on the output only if both inputs are also true. The second gate is a XNOR gate that produces true on the output whenever both inputs are equal.

Both gates of the circuit in Figure 2.1, as well as their combined output, can be represented to individual truth tables as shown in Table 2.1. The combined truth table is a three-input, one-output table. In the FPGA world, such a truth table is called a Lookup Table (LUT). All logic functions that can be expressed can also be represented in the form of a LUT which makes them the perfect basic building blocks for FPGAs. Nowadays, the LUTs employed in FPGAs are much bigger than three-input, one-output. A typical configuration nowadays is six-input, one-output. Typically, those LUTs are implemented as very fast SRAM cells which lose their state after power down. Some FPGAs use Flash to

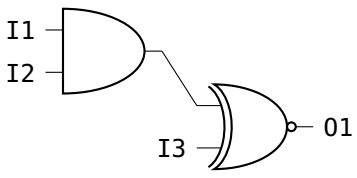


Figure 2.1: Simple three input, one output logic function. An AND gate is connected to a XNOR gate.

I1	I2	O1
0	0	0
0	1	0
1	0	0
1	1	1

(a) AND

I1	I2	O1
0	0	1
0	1	0
1	0	0
1	1	1

(b) XNOR

I1	I2	I3	O1
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	1	1

(c) Combined

Table 2.1: Truth table derived from the schematic in Figure 2.1

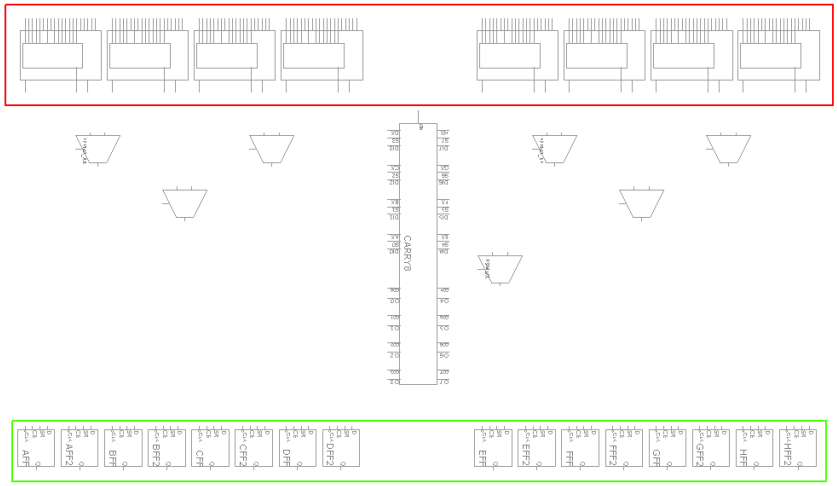


Figure 2.2: View generated by Vivado 2019.1 of the basic logic blocks of a Xilinx Virtex UltraScale+ device. The CLB contains eight six-input, one-output LUTs (marked in red) and sixteen FFs (marked in green). CLBs are the basic block of the Virtex UltraScale+ architecture.

avoid an empty device during bootup at the cost of performance and efficiency. To improve flexibility such LUTs can typically be combined to form larger LUTs or split into separate functions depending on the use case.

The second resource any modern FPGA has in abundance are Flip-Flop (FF). FFs are single bit state elements that save their input on the rising edge of a clock signal. Their purpose will be explained further down in Section 2.4.

So how does the basic building block containing the LUTs and FFs on a modern FPGA look like and what other components does the block contain contain? Figure 2.2 shows a so-called Configurable Logic Block (CLB) as generated by Vivado 2019.1 as an implementation overview for an Alveo U250 device. Marked in red are the eight

individual LUTs. Each of the LUTs is highly configurable and supports the following modes as specified in *UltraScale Architecture Configurable Logic Block User Guide (UG574)* [126, p. 16]:

- "Any arbitrarily defined six-input Boolean function."
- "Two arbitrarily defined five-input Boolean functions, as long as these two functions share common inputs."
- "Two arbitrarily defined Boolean functions of three and two inputs or less."

Accordingly, a single CLB can serve as eight six-input LUTs, as sixteen five-input LUTs (as long as each pair of two share common inputs) or any combination of the two. The LUTs can also be used as so-called multiplexers, a type of circuit that forwards a certain input port to the output depending on the state of a set of selection bits. These multiplexers can be used to route signals inside of the CLB.

Marked in green are the 16 FFs which will be explained in Section 2.4. Additionally, the CLB contains a variety of functionalities that can be used to efficiently implement certain designs. For example a dedicated carry-logic can be used to implement arithmetic operations very efficiently. Lastly, the LUT can be used as Random-Access Memory in contrast to their usual behaviour as Read-Only Memory.

By using a CLB any logic function fitting in the resources can be represented. However, usually the logic function, for example a $32\,\mathrm{bit} \times 32\,\mathrm{bit}$ multiplier, is much bigger than a single CLB can fit. Accordingly, there has to be a way to connect different CLBs up in a flexible and configurable manner.

2.2 CONNECTING THEM UP

Note the difference between LB, a general logic block, and the CLBs, which are a specific implementation of a LB, used in Virtex UltraScale+.

The basic principle of FPGA architectures should be clear after reading the previous section. However, a single LUT is not very powerful on its own. To build larger designs there has to be a way to chain different LUTs, or in the case of the Virtex UltraScale+ architecture, CLBs together.

Let's take a look on a island style FPGAs architecture as a specific example in Figure 2.3. The logic blocks (LB) are organized in a grid across the entire FPGA. Located between any of those LBs a so called connection box (CB). The purpose of the CBs is to connect the LBs with the routing grid that runs horizontally and vertically across the FPGA. The routing grid itself is switched inside switch boxes (SBs).

Traditionally, FPGA vendors treat the switching infrastructure as one of their most valuable trade secrets. What little is known stems from educated guesses, reverse engineering and old designs.

The implementation, and accordingly the configurability, of those SBs varies a lot between different FPGA architectures, the fundamental purpose, however, is the same for all implementations. The SBs offer a certain number of inputs and a certain number of outputs. Based on their configuration they connect certain inputs to certain outputs. Typically, many constraints apply to SBs which are a result of their

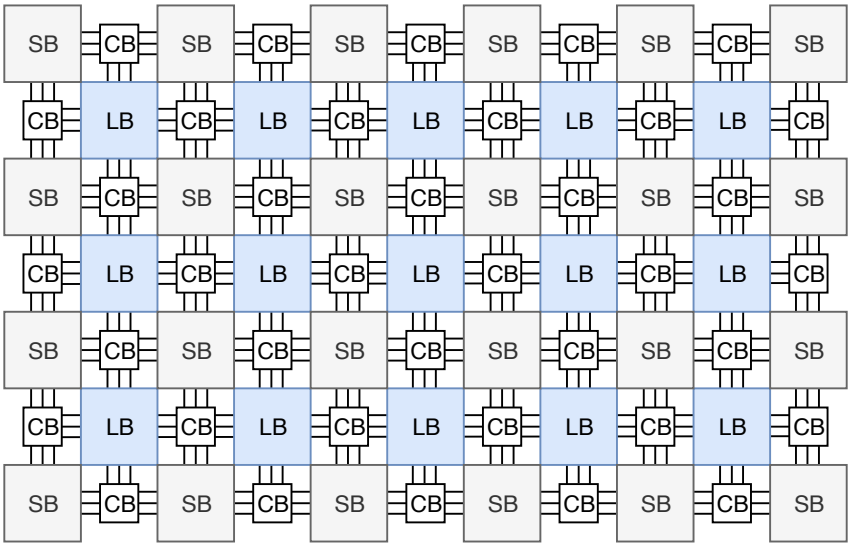


Figure 2.3: Simple switching architecture in a island style FPGA [71]. Tracks are layed out horizontally and vertically across the die space. The inputs and outputs of logic blocks (LBs) are connected with the routing network via connection boxes (CBs). The individual traces in the routing network can be connected with one another in switch boxes (SBs).

(highly efficient) implementation and increase the routing difficulty. For example a certain type of SBs could only connect inputs and outputs in a certain direction: top to bottom, left to right and top to right are valid, but not left to right and right to left at the same time.

Switch boxes (SB) are the fundamental switching resources in FPGA. They come with certain limitations on their routing directions and connectivity.

The basic design of a highly regular switch box grid around a logic block grid can be improved in a variety of ways. A typical extension is the introduction of a hierarchy in the SBs to speed up the connection between LBs located further apart. The hierarchies are also often coupled with certain "high-speed" connections with very high bandwidth to route across larger distances on the FPGA. Other extensions include dedicated routing resources for special signals.

In modern FPGA architecture there is an abundance of routing resources. Hence, a lack of these resources seldom constrains the design. However, there is certainly no unlimited number of connections available and with designs that are very constrained to a specific location, such as memory controllers, routing can become an issue.

Logic blocks and switch boxes are enough to build a working FPGA. Some special resources for interaction with the outside world, usually referred to as Input-Output- (IO-)Blocks have to be added. However, over the years FPGA vendors used the available die space for a variety of additional resources.

2.3 ADDITIONAL INFRASTRUCTURE

The basic FPGA described up till now is already capable of implementing any logic function. However, certain operations are frequently used and would require a lot of logic resources. Modern FPGAs contain special units that handle only the specific function and can be connected via the routing network with the rest of the programmable logic.

Specialized resources for memory and arithmetic operations are very common. Additionally, there are often specialized units for high speed IO applications such as PCIe.

One of the most fundamental and often used blocks are dedicated memory blocks. Modern logic blocks can of course be configured to act as random-access memory, nonetheless, they offer only limited space and reduce the amount of logic that can be put on the device. Accordingly, a modern FPGA contains many Block-Random-Access-Memories (BRAM) that act as very fast SRAM and can be hooked up to the user logic on the FPGA. The feature set of the BRAMs varies depending on the architecture, but at the very least, they support single port access to memory. BRAMs can usually be configured for different sizes and input width, for instance, $36\,\mathrm{bit} \times 512$ or $72\,\mathrm{bit} \times 1024$. The amount of BRAM available depends on the architecture but usually there is a high availability. Modern architectures employ a tiered approach and add additional layers of slower but larger RAM, such as the aptly named "UltraRAM" of Virtex UltraScale+.

The second almost universally available resources are specialized for arithmetic operations. They usually can handle operations such as additions and multiplications for a certain bit width such as $16\,\mathrm{bit} \times 16\,\mathrm{bit}$. Larger operations can be realized by chaining the operators. Next generation FPGA such as Xilinx Versal provide new resources in this area by providing dedicated vector processors for high performance numerical operations.

There are other dedicated resources, but memory and arithmetic blocks are most commonly used. One additional resource worth mentioning in this context are dedicated high performance IO units. These units can be used to interface with interconnects such as PCI-Express 4.0 or $100\,\mathrm{Gbit/s}$ Ethernet.

Modern synthesis tools are capable of finding suitable specialized blocks based on the input source code through a process called inference. Alternatively, the blocks can be instantiated directly in the source code if certain architecture dependent functionality is necessary. Direct instantiation comes with the penalty of vendor and architecture lock-in which is in most cases not desirable (Chapter 1 for a brief discussion of porting issues).

2.4 THE SPEED OF LIGHT

With the information provided in the preceding sections one could start building infrastructure for FPGA. For example, a big multiplier

FPGA are often used as one of the first designs on a new process node. Modern FPGA are produced using between 16 nm and 7 nm processes.

to operate on 64 bit floating point numbers can be implemented by using the LUTs to represent the logic of everything that is necessary. In this example, the necessary logic needs to (1) add the exponents, (2) multiply the mantissas and (3) normalize the results. This behavior can be built from some basic multiplication logic based on the repeated addition known from school and full adders for addition. Sadly, the reality, and especially physical laws, come in the way of such concepts. Even the tiny circuits in today's FPGAs are limited by the speed of light. No matter what technology is used, the electrons have to move at a speed lower than the speed of light between transistors. Accordingly, the design of our multiplier will have a very large propagation delay. The propagation delay is the time it takes until an output reflects the changes of the corresponding input. For large circuits that delay can be surprisingly large, ranging into the seconds.

Hence, small circuits with state elements inbetween are a way to limit the propagation delay. These smaller circuits have a short propagation delay. The state elements break the path up into smaller pieces. All of the small circuits are then controlled with a shared time step. The time step, called a clock, has to be chosen long enough that all the possible paths through the small circuits are considered. The state element commonly chosen are FFs. As those little memory elements, usually referred to as registers, are needed between all of the tiny circuit elements, they are also available in big quantities. Going back to the FPGA introduction: A CLB has eight LUTs but 16 FF. The logic, previously referred to as small circuits, is called combinatorial logic.

There is a different approach called asynchronous circuit design. Instead of a clock, careful path length calculations and handshakes are used to make sure the calculation is correct. A large circuit is built up from FF with combinatorial logic in-between. The longest path between two FF through the combinatorial logic determines the clock period of the design.

In detail there are a lot of problems introduced by this model. For example, the clock period has to be precisely chosen to be just long enough for all paths of a circuit to execute. Any shorter and a certain change on the input might not be reflected on the output and the result of the calculation is wrong. Accordingly, finding the longest, or critical, path is a fundamental problem for the CAD tools targeting FPGA. Typically the designer of the circuit gives certain constraints about the clock period. For instance, the clock period is constrained to be 5 ns. The Place&Route process then has to set out and find a configuration that meets this constraint. For some designs the CAD tools have no chance to find a successful configuration. The designer might require two Digital Signal Processor (DSP) cells between two FFs to form a multiply-accumulate operation. However, assume a single run through the DSP takes 3 ns. Without even accounting for the length of the wires used for routing between the DSPs it is already obvious that the target of 5 ns cannot be reached. The tool will fail timing and the designer has to rethink the circuit. The problem can be resolved by introducing an extra pipeline register, which is a FF between the chained DSP cells. The critical path now passes only one DSP unit instead of two and might be able to pass timing. Of course this change is not free and requires an additional FF and additional routing

resources. For larger designs those changes are not trivial and might have unexpected consequences, especially for very densely packed circuits where almost all of the routing resources are already utilized. Another frequently faced challenge is the selection of dedicated units. The dedicated DSP might be a lot faster than the same circuit built up out of LUTs. However, the routing paths to and from the DSPs might already be too long and the LUT based approach might result in a lower critical path.

So far, this chapter detailed the most important concepts of targeting FPGA:

- Fundamentally, FPGA contain LUTs to represent logical functions and FFs to store state.
- Additional resources are available for special applications such as memory or arithmetic operations.
- The execution is controlled by a common clock that makes sure that the propagation delay through the combinatorial logic is controlled.

The rest of this chapter briefly introduces the tool side of FPGA targeting.

2.5 TARGET MAPPING

Nothing stops a user from configuring FPGA resources directly. Using the LUT primitives and storing the state in individual FF makes up the circuit. Afterwards the designer selects which virtual LUT is mapped onto which physical LUT of the target design. A vendor provided tool creates the necessary configuration files, called a bitstream, to program the FPGA. Surprisingly, there are certainly designers that try to approach the limits of FPGA design using this technique to achieve ultimate control of the device. For example, [43] tries to fit as many functional RISC-V processor cores onto a single FPGA as possible. However, just as with assembler programming on CPUs, this approach is very error prone, tedious, and nowadays not common. Instead there are specialized tools that take a higher-level representation of a design and translates into the correct configuration for the FPGA target.

The first step is the synthesis of the design. It takes an abstract textual representation, usually VHDL or Verilog, and determines which virtual FPGA resources of the targeted architecture are necessary and how they should be connected. For example, the user might write a multiplication of two numbers into the source code. The synthesis process will then determine how many and what type of LUTs are necessary to form this multiplication. The synthesis process will also remove any unnecessary parts, for example signals that never change.

Synthesis translates from mostly target independent abstract representations of a circuit to a target specific and optimized version of the same circuit.

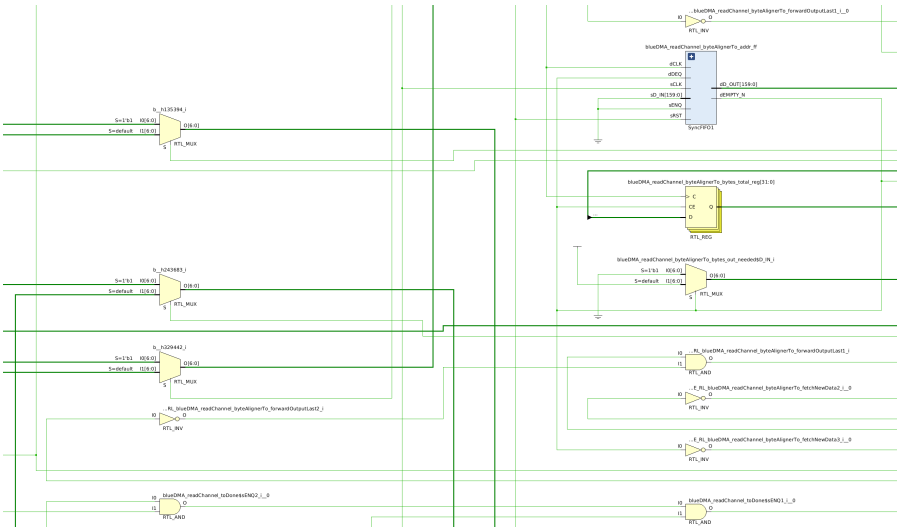


Figure 2.4: Part of a Schematic generated by Vivado 2018.3 after synthesis. The yellow boxes represent the individual primitives such as LUTs or FFs of myriad configurations. The blue boxes contain instantiated modules that can be reused across the schematic. The schematic represents part of a circuit used for copying data from one memory location to another.

Modern synthesis tools can go one step further and also optimize certain aspects of a design. Most frequently encountered is the state machine detection that finds state machines in the abstract representation and finds a well-suited state name scheme depending on the target architecture, often minimizing multiplexers as those are rather expensive.

Additionally, a synthesis tool is capable of finding structures in the abstract design that can be represented by dedicated units of the target architecture such as DSPs. This process, called inference, allows writing target independent code which does not have to instantiate target specific cells.

A graphical representation of a synthesized design as generated by Xilinx Vivado is shown in Figure 2.4.

The output products of the synthesis can then be used to place the virtual cells on the physical FPGA in a process called Place&Route explained hereafter.

2.6 PLACE&ROUTE

After synthesis the tool already knows what kind of cells are most likely needed to build the circuit and how they should be connected. However, the correct placement of those cells onto the physical FPGA is not known. To reach the final bitstream used to configure the FPGA, the mapping of the virtual cells to the physical cells has to be generated. This process is called Place&Route.

Place&Route, the process of mapping a synthesized design onto a physical FPGA, is usually very slow and can take multiple days to complete.

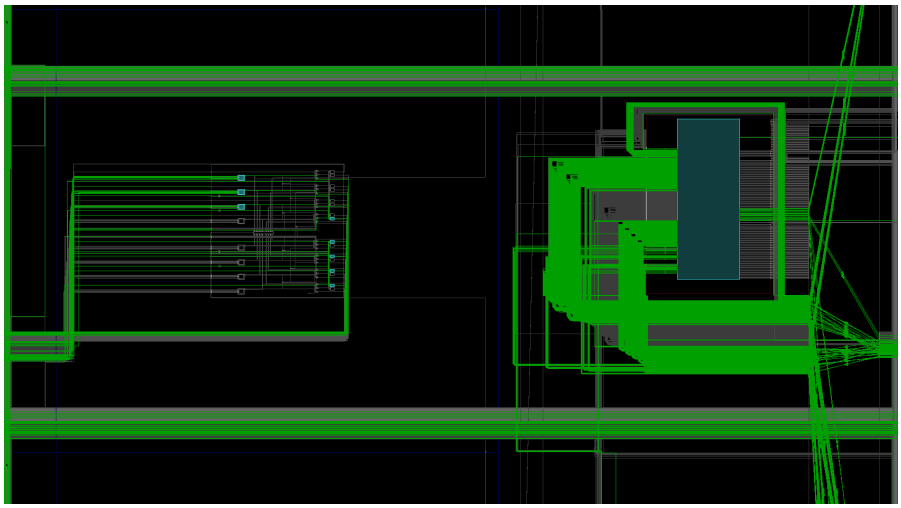


Figure 2.5: Graphical representation of a Alveo U250 FPGA after Place&Route as generated by Vivado 2018.3. The blue boxes are those primitives selected by the Place&Route process. The green lines represent the connections between those primitives. The FPGA is very regular with lines containing the same resource such as CLBs, including LUTs and FFs (the small blue boxes), or BRAM (the long blue box).

Sadly, as hinted at in Section 2.4, this process is not simple. There is an unfeasibly large number of mappings available between the virtual and the physical cells. Additionally, certain constraints apply that makes finding a working configuration even harder. Most prominently, those are timing constraints that force the design to meet certain clock periods. Additionally, certain placement constraints limit the available physical cells for a given virtual cell, based for example on IO pads of the FPGA. As the solution space is vast, finding a solution deterministically is often not possible. Even worse: When starting the process, it is not even clear if any solution can be found. Accordingly, in the real world the designer might experience a failed run after coming back to the office after the weekend as the placer is not able to find a mapping that meets timing, or even worse, does not find a mapping at all. In the latter case, the designer has to go back to the drawing board and reduce resource usage. In the former case, the designer might introduce additional pipeline stages or rerun the process with less tight timings.

This concludes the coarse description of the structure of a FPGA. In detail there are a lot more things to consider. For example, it might be better avoiding DSPs and build up certain arithmetic functions using LUTs. This is the case when operations do not map well onto DSPs or the routing penaltiy to reach the dedicated block is too high. The next chapter will describe common HDL used in today's academia and industry to generate the designs that go through the process mentioned above.

3 METHODS OF HARDWARE DESCRIPTION

Chapter 2 introduced the hardware side of FPGAs. However, languages are needed to efficiently design hardware for it. The days of HDLs as the only choice are gone and a wide selection of different languages is available. This chapter gives an overview of many of the popular and some of the less known languages for hardware design. Most of the languages presented in this section are not FPGA specific and can also be used to target ASICs. In contrast to HLSs, presented in Section 3.4, a HDL directly describes the structure and behavior of hardware at a certain level of abstraction.

This section is not a complete introduction to hardware design using HDL. Other sources such as [11, 44] are better suited for teaching those languages. Instead, this chapter is designed to give an overview of how the languages abstract hardware design and how they look like. For this purpose a common example design will be implemented in all of the presented languages. Additionally, exerts from the generated Verilog of the HCLs will be shown for comparison.

3.1 SCENARIO

The comparison is done using a circuit that shows fundamental differences between the languages. A common functionality which is used in many applications is a Finite Impulse Response (FIR) Filter. An incoming discrete signal is filtered by applying a filter function to the last n samples and summing them up to produce the new output signal. The generalized approach is shown in Figure 3.1.

The goal of each implementation in the different languages is to implement a FIR Filter with five taps for 16 bit signed integers. If

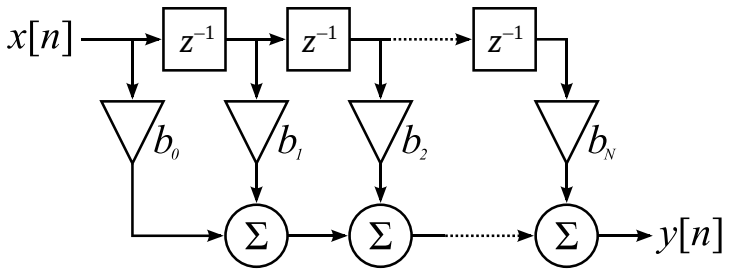


Figure 3.1: FIR Filter of order n. The figure is available under public domain[14]. The incoming signal is delayed to be used in later iterations. For each output signal $y[n]$ the current input sample, as well as the last four input samples, is multiplied with their corresponding coefficient and summarized.

possible, a generalized FIR filter should be implemented including a specialized four tab implementation. The chosen filter weights are 1, 2, 3, 2, 1. These settings are chosen to provide, on the one hand, easy to follow implementations, while on the other hand, still remain complex enough to show language differences.

The FIR filter is a non trivial example of a circuit that is frequently used and can be implemented in most HDLs and in HLS.

All interfaces use a so called handshake mechanism to synchronize interface activation. The handshake consists of two signals, a ready and a valid signal. The valid signal is set by the party that controls the data and the ready signal is set by the receiving party. For instance, the outside world that wants to input a sample into the accelerator controls the valid signal. The accelerator itself controls the ready signal to indicate readiness to receive new data. Whenever both signals are high at the same time the corresponding operation is executed. Handshake mechanisms are frequently encountered in todays circuit design and are a fundamental piece of the global asynchronous, locally synchronous method of circuit design. The handshake is used to allow both partners in the communication to control the flow of the communication. If, for example, the slave is busy and can not take a new request it indicates that by setting the corresponding ready signal to low.

Remember that the circuit is synchronous and the signals are only registered on the rising edge of the clock.

The rest of this chapter is split into two parts. Initially, the classical HDLs are introduced. In the second part of this chapter so called HCLs are presented as a novel approach to hardware design without resorting to HLS.

3.2 CLASSICAL HDLS

For a long time there were mainly two players in the HDL game: VHDL and Verilog. The former was constructed around 1983 to document ASICs and is heavily influenced by Ada [7]. Verilog came a bit later in 1984 and follows a more C like syntax [94]. Both languages are usually used at the Register-Transfer Level (RTL) and can usually be transpiled into one another [33]. Apart from designing circuits using the languages, verification is also supported and both languages can be used to build testbenches. Their low level of abstraction makes either language a good candidate to be used as a common denominator for tool support. The HCLs described later in this chapter usually compile their code into either language.

Verilog and VHDL are widely used despite being akin to assembler in software programming.

In detail the languages have certain features which make them feel quite different despite being equally powerful. VHDL is said to be rather verbose and performs additional error checking. In VHDL a wire that is not connected leads to warnings and errors further down the toolflow. Verilog on the other hand is less strict and does not enforce many of the rules that are required by VHDL. Both languages come with certain caveats based on their origin as system verification languages. Accordingly, they contain constructs that can not be trans-

There is a brisk debate on which language is the superior one, as it is often the case with two competing alternatives and engineers with too much time.

ferred into hardware as they are only applicable to simulation, which is often a nuisance for beginners trying to get into hardware design. Usually the circuits simulate fine but the tools used for synthesis are not able to work with the code or build undesirable hardware representations.

Nonetheless, for many years both languages shared the circuit design market and are still heavily used to this day. VHDL mainly stayed true to its roots with minor adjustments along the way. Verilog received a layer of abstraction on top to simplify design verification known as SystemVerilog.

As both languages have basically the same level of abstraction, the example code in Listing 3.1 shows only the Verilog implementation of the circuit described in Section 3.1.

The code closely follows the original circuit. Despite using some generalized concepts such as loops to replicate hardware, no higher level abstraction is possible. Generalization mainly happens through parameters that can be used to define different widths of the registers or additional filter tabs. The fundamental functionality, however, is hard wired and requires more detailed changes to be adopted to new applications.

Things such as the handshake semantics or the valid bits for the output have to be described explicitly. There is no type system in place and the semantics of the individual registers are completely open to the developers discretion.

Over the years new languages emerged that aim to enhance HDLs by adding high level features or different design methodologies. These languages are still HDLs at heart but the term is usually associated with Verilog and VHDL. The term HLS on the other hand does not fit properly either, because the languages used for HLS do not directly represent hardware and are usually DSL or re-purposed software programming languages. The authors of Chisel coined the term HCL for their language to avoid the misinterpretation of the traditional notations.

3.3 HARDWARE CONSTRUCTION LANGUAGES

Hardware Construction Languages (HCLs) are a subset of HDLs that aims to improve on the traditional HDLs representatives in a number of ways. Typical for HCLs are a type system and a higher level of abstraction. For instance, much of the functionality for synchronization such as handshakes can be implicitly handled.

HCL should not be confused with HLS. Whereas HCLs directly model hardware, be it at a high level of abstraction, HLSs use languages without any relation to hardware design and infer the circuits from the logic of the languages.

The following section presents two of the most popular HCLs:

```
module fir (
  input clk,
  input rst_n,

  input [15:0] x,
  input x_valid,
  output x_ready,

  output [15:0] y,
  input y_ready,
  output y_valid
);
// The filter coefficients
localparam [15:0] coeff0 = 1;
localparam [15:0] coeff1 = 2;
localparam [15:0] coeff2 = 3;
localparam [15:0] coeff3 = 2;
localparam [15:0] coeff4 = 1;

integer i;

reg [15:0] unfiltered[4:0];
reg [4:0] unfiltered_valid;

reg [15:0] unfiltered_next[4:0];
reg [4:0] unfiltered_valid_next;
// Take new values at the clocks positive
↪ edge
always @ (posedge clk) begin
  if(!rst_n) begin
    for(i = 0; i < 5; i = i + 1) begin
      unfiltered[i] <= 0;
      unfiltered_valid[i] <= 0;
    end
  end else begin
    for(i = 0; i < 5; i = i + 1) begin
      unfiltered[i] <=
      ↪ unfiltered_next[i];
    end
    unfiltered_valid <=
    ↪ unfiltered_valid_next;
  end
end
// Calculate new register values
always @ (*) begin
  // Start with the value of the last
  ↪ cycle
  for(i = 0; i < 5; i = i + 1) begin
    unfiltered_next[i] = unfiltered[i];
  end
  unfiltered_valid_next =
  ↪ unfiltered_valid;

  // Output has been cleared
  // -> Remove valid from last sample
  if(y_ready && y_valid)
    unfiltered_valid_next[4] = 0;
  // Shift value if the next register is
  ↪ empty
  for(i = 4; i >= 1; i = i - 1)
  begin
    if(unfiltered_valid_next[i - 1]
      && !unfiltered_valid_next[i])
    begin
      unfiltered_valid_next[i] = 1;
      unfiltered_valid_next[i - 1] = 0;
      unfiltered_next[i] =
      ↪ unfiltered_next[i - 1];
    end
  end
  // Take new input value if a register
  ↪ is available
  if (x_ready && x_valid)
  begin
    if(!unfiltered_valid_next[1])
    begin
      unfiltered_valid_next[1] = 1;
      unfiltered_next[1] = x;
    end else begin
      unfiltered_valid_next[0] = 1;
      unfiltered_next[0] = x;
    end
  end
end
// Calculate output based on coefficients
assign y = unfiltered[0] * coeff0
         + unfiltered[1] * coeff1
         + unfiltered[2] * coeff2
         + unfiltered[3] * coeff3
         + unfiltered[4] * coeff4;
// Output is valid if all registers are
↪ valid
assign y_valid = &unfiltered_valid;
// Input is ready if first register is
↪ empty
// Or the output is cleared in this cycle
assign x_ready = !unfiltered_valid[0]
               || (y_valid && y_ready);
endmodule
```

Listing 3.1: Verilog implementation of the circuit described in Section 3.1. The implementation is optimized to take inputs every cycle when available. For instance, if a value is take from the FIR filter via the y-output, a new value can be taken through the x-input in the same cycle. Features of higher level languages such as a proper type system or higher-order-functions are not available. All the necessary semantics, for example for the handshakes, have to be enforced manually.

- Chisel: The language which coined the term HCL. Provides a higher level of abstraction compared to Verilog or VHDL and is built upon a strong type system. The basic thought process during development does not differ from the classical HDLs. Chisel describes the circuits structure.
- Bluespec: Language with a completely different concept. The introduction of the concept of "guarded atomic actions" shifts the way hardware design is done while still maintaining very fine grain control over the resulting circuits compared to HLSs. Bluespec describes the structure of the circuit but can also describe the behavior of it.

3.3.1 *Chisel*

Chisel is a DSL embedded in Scala proposed in [10]. It is used to target ASIC and FPGA flows through RTL descriptions and, in that regard, has the same general direction as the classical HDLs. So how does Chisel differ? Chisel is based on the very powerful Scala programming language and extends the base language with constructs necessary for hardware design. In that regard, Chisel tries to avoid certain pitfalls found in classical HDLs. For instance, HDLs were invented as simulation focused languages and contain constructs that can not be synthesized. Hence, a developer has to know which constructs are valid for synthesis. Chisel makes the distinction between the two options explicit. Nonetheless, Chisel offers very powerful methods for simulation and verification.

The general level of abstraction in regards of hardware design stays the same compared to Verilog and VHDL. However, Chisel offers features that simplify the lives of hardware developers. Most of the features of the base language can be exploited to generate complex structures automatically. For example, a FIR filter can be built on a base unit that takes a list of functions which will be applied automatically to the last n samples. The filter functionality is then separated from the structure of the delay circuit to have the right sample at the right time.

This is certainly only a simple example. Chisel offers many more advantages such as a proper type system and correctness checks through the compiler. All features from Scala can be used for testing and debugging, such as automatic testing frameworks for test stimulus generation. Chisel offers a simulation engine directly in Scala which can be faster than a full blown simulation based on Verilog. The initial verification and testing can happen directly in Chisel. The finalized design is then compiled to Verilog and can be processed by a normal Verilog based tool-flow.

The FIR Filter example in Listing 3.2 looks quite different than the one in Verilog and increases flexibility. Instead of building a FIR filter

for a set number of tabs, a generalized FIR filter is implemented and then the coefficients are simply fed into the module during initialization.

The interface of the module uses the `Decoupled` type to implement the handshakes. The handshake semantics are still handled by the programmer.

Instantiation of the triangle filter is done by

```
1 val triangleFilter = FirFilter(16.W, Seq(1.U, 2.U, 3.U, 2.U, 1.U)).
```

The same module can be reused to build a wide variety of filters. A simple two cycle delay, for example, could be created using

```
1 val delayFilter = FirFilter(16.W, Seq(0.U, 0.U, 1.U)).
```

3.3.2 *Bluespec*

Bluespec is a Haskell based HDL that originates at MIT [90]. During the early years the syntax was closely intertwined with Haskell. To help the increasing commercialization with the founding of the Bluespec company, the syntax has been changed to resemble SystemVerilog. The change promised increased appeal in the wider hardware design community. However, comparing it to SystemVerilog would be unfair. Bluespec introduces completely new concepts that can not be found in any other language and bring a totally different thought process into architecture based hardware design.

Bluespec is a proprietary HDL with a new twist to hardware design: guarded atomic actions.

The language is built on "guarded atomic actions". Multiple actions can be grouped in "rules". Each rule contains actions that have to be executed atomically, so, at the same time. Each action might have certain limitations on when it can be executed, for example, putting data into a `FIFO` is only possible if the `FIFO` is not already full. These conditions are called implicit guards. Additionally, each rule can be annotated by explicit guards that inhibit the firing (execution) of the rule. A rule might only fire when a certain counter value is reached, for example.

This simple concept is incredibly powerful when it comes to hardware design. The compiler takes the rules and tries to schedule as many rules as it can for parallel execution. The rules might be in conflict to one another - a `FIFO` can only take one element per cycle. If two rules want to access the `FIFO`, the compiler or the developer has to decide which rule can fire and which has to wait for the other rule.

Interfaces to other modules use handshakes by default which promotes latency insensitive design. However, the behavior can be changed through compiler attributes. The compiler will then ensure that the desired behavior is enforced throughout the modules. If an interface is specified as having no handshake, the compiler will throw an error if the interface is not read or written (depending on the direction) in every clock cycle. Thanks to the strong type system, architectures that

```
class FirFilter(bitWidth: Int, coeffs: Seq[UInt]) extends Module {
  val io = IO(new Bundle {
    val in = Flipped(Decoupled(UInt(bitWidth.W)))
    val out = Decoupled(UInt(bitWidth.W))
  })
  // Create the serial-in, parallel-out shift register
  val rdy =  Reg(Vec(coeffs.length, Bool())) // Stores the valid entries
  val zs = Reg(Vec(coeffs.length, UInt(bitWidth.W)))

  // Shift all values forward that are valid, if the next register is
  ↪  empty
  for (i <- 1 until coeffs.length) {
    when(!rdy(i) && rdy(i - 1)) {
      zs(i) := zs(i-1)
      rdy(i) := true.B
      rdy(i - 1) := false.B
    }
  }

  // Do the multiplies
  val products = VecInit.tabulate(coeffs.length)(i => zs(i) * coeffs(i))

  // Input handling
  // By default, the input is not ready
  io.in.nodeq()

  // If the first slot is empty
  when(!rdy(0)) {
    // Be ready to accept a value
    io.in.ready := true.B
    // Accept the value if it's valid
    when(io.in.valid) {
      // Store value in first stage of shift register
      zs(0) := io.in.bits
      rdy(0) := true.B
    }
  }
  // Output handling
  // Sum up the products
  io.out.bits := products.reduce(_ + _)

  // If all elements used for output computation are valid
  when(rdy.reduce(_ & _)) {
    // Signal that the output is valid
    io.out.valid := true.B
    when(io.out.ready) {
      // Remove last element, as the output has been read
      rdy(coeffs.length - 1) := false.B
    }
  }.otherwise {
    io.out.valid := false.B
  }
}
```

Listing 3.2: Chisel implementation of the triangle response FIR filter. The filter is implemented for any number of coefficients and automatically adjusts. The desired configuration is chosen during instantiation.

are often difficult to write in a correct manner in traditional HDL are very easy to do. Having multiple clocks in a design is usually hard to do as the designer has to ensure that any signals that cross the clock domain are properly handled. In Bluespec, the compiler can ensure that those crossings are handled as the different clocks are of different types.

Bluespec can interface with other HDLs by defining the interfaces of VHDL or Verilog modules as their Bluespec equivalent.

The Bluespec compiler can generate a variety of output formats. The Bluespec simulator runs natively on the host system and is based on C++. The simulation is orders of magnitude faster than HDL based simulation. Additionally, Verilog can be generated and the standard Verilog based tool-flows can be used. Lastly, SystemC can be generated to be used in verification. The tool-chain is very mature and efficient. The proprietary nature of the compiler, however, is detrimental to its success outside of certain companies and universities.

The FIR example written in Bluespec and presented in Listing 3.3 comes with much of the convenience already experienced with Chisel. For instance, the number and value of filter coefficients are configurable by providing a list of them. However, there are certain aspects that can not be done as easily with Chisel. The rules encapsulate a lot of the logic that is required to, for example, ensure handshake mechanisms. Line 26 defines the rule `output_sum` which does not have any explicit guards. Accordingly, one could think that this rule is scheduled every cycle. However, this would not make sense from a functionality standpoint as the sum can only be calculated if enough data is available and the previous value has been taken from the output FIFO. Hence, the rule is guarded by implicit guards. There are certain more obvious ones: (1) the last FIFO of the `stages` list has to contain an element to call `deq` on it, (2) the `out` FIFO has to be able to take another value to call `enq` on it. Additionally, there are some less obvious ones. The sum functionality calls `first` on the FIFOs in `stages`. This call is only allowed if there is an element in the corresponding FIFO. Accordingly, this two line rule has $n + 2$ guards when n is the number of coefficients.

The standard library of Bluespec contains many useful operations. As an example, the `fold` function employed in line 22 results in a binary tree fold. There are other versions of this functions, like the `foldl` and `foldr` functions which perform the operation linearly from left or from right over the list.

Instantiating the module is done by providing the desired coefficients during module instantiation. The triangle impulse example becomes

```
1 let triangle <- mkFIR(cons(1, cons(2, cons(3, cons(2, cons(1, Nil))))));
```

```
module mkFIR#(List#(Int#(16)) coeffs)(Server#(Int#(16), Int#(16)));
    FIFO#(Int#(16)) in <- mkPipelineFIFO();
    FIFO#(Int#(16)) out <- mkPipelineFIFO();

    // Store size of list to have nicely named reference
    Integer num_coeffs = length(coeffs);

    // Each coefficient receives one FIFO element
    List#(FIFO#(Int#(16))) stages <- replicateM(num_coeffs,
    ↪  mkPipelineFIFO());

    // Helper function to be used in higher order functions.
    // Returns first element of a FIFO without removing it from the
    ↪  FIFO.
    function a getFirst(FIFO#(a) f);
        return f.first();
    endfunction

    // Multiply coefficients with the corresponding value
    // and build a sum using a binary tree (fold)
    function Int#(16) sum();
        List#(Int#(16)) vals = map(getFirst, stages);
        let m = zipWith(\* , vals, coeffs);
        return fold(\+ , m);
    endfunction

    // Put output into sum and drop last element
    rule output_sum;
        stages[num_coeffs - 1].deq();
        out.enq(sum());
    endrule

    // Connect all other stages
    for(Integer i = 0; i < num_coeffs; i = i + 1) begin
        if(i == 0) begin
            mkConnection(toGet(in), toPut(stages[0]));
        end else begin
            mkConnection(toGet(stages[i - 1]), toPut(stages[i]));
        end
    end

    interface request = toPut(in);
    interface response = toGet(out);
endmodule
```

Listing 3.3: FIR Filter example implemented in Bluespec. Coefficients are provided as a list of integers. The infrastructure for the filter is built up based on this list. Bluespec provides many high level functions to simplify writing hardware such as the `fold` function for binary tree operations, or `zipWith` and `map` known from programming languages.

3.3.3 *Other HCLs*

Chisel and Bluespec are certainly the most commonly used among the HCLs, but not the only ones. Another notable HCL is the Python based MyHDL[31]. The language is neither integrated into a base language, nor defines a completely new language like Bluespec. MyHDL comes along as a normal Python library. It extends beyond VHDL and Verilog by utilizing some of the features of Python to provide better generalization. However, the main parts of the program still look very similar. The language adds features such as proper type support and differentiation between simulation and synthesizable code.

The authors of MyHDL are very outspoken when it comes to defending their language against newer competitors such as Chisel.

Next to Chisel other Scala based HCLs exist such as SpinalHDL [95]. In general they follow the same approaches and are very similar in syntax to Chisel and profit from the very high flexibility of Scala as a base language for embedded DSL.

Another few languages are built based on Haskell, similar to Bluespec. Compared to Bluespec they usually follow an approach closer to that of Chisel and are realized as embedded DSL compared to Bluespec which is its own language. A notable example is Clash [9] which takes full advantage of Haskells powerful inference and type system to automatically detect errors and deduce structures.

As the fundamental concepts of these languages do not differ essentially from the previously introduced examples, the FIR filter example is not extended to them. Another approach, however, is taken by HLS, where hardware is not directly described but deduced from a functional description.

3.4 HIGH LEVEL SYNTHESIS (HLS)

Both HDL and HCL have a steep learning curve due to their nature as hardware descriptions. The thought process going into a circuit design is not easily comparable to designing software. Especially for FPGA vendors this poses a big marketing problem in the accelerator space. Programming multi-threaded CPUs and GPUs is very straight forward and largely comparable to one another. Unleashing the acceleration capabilities of FPGA, however, requires the developer to learn completely new languages with totally different characteristics. Stemming from this need, all major FPGA manufacturers developed so called HLS tool-flows aiming at software developers without major hardware knowledge.

The circuit is described not by its inherit behavior but rather through the desired functionality. A compiler then takes the description and tries to deduce a working circuit with the expected functionality. For general purpose programming languages, this is no trivial task and many generations of scientists and engineers alike are working on this still open issue. Details such as proper scheduling of tasks over the

given resources, introduction of register boundaries to form pipelines or handling of recursion and loops make HLS a very demanding topic for compiler engineers.

This chapter introduces three different methods of HLS that are commonly found in the industry and research. Talking about HLS usually refers to synthesizing hardware from general purpose programming languages such as C and C++. Being the most common approach, it is largely supported by many tools. A different approach is taken by DSL based HLS. Instead of supporting general purpose programming languages with the accompanying problems, specialized languages for a narrow domain are used as basis for synthesis. Common examples of domains include machine learning and bioinformatics. The narrower set of language features improves the synthesizeability and improves the performance of the resulting circuit. Additionally, SystemC, a mixture of HCL and HLS approaches, is presented. Still following many features found in HLS, SystemC also supports description of hardware details such as clocking behavior.

3.4.1 *Programming Language Based HLS*

The dream of synthesizing hardware directly from software implementations is certainly not a new one. The advantages are clear: Software that turns out performing poorly on general purpose hardware can simply be compiled to run directly in hardware with a high increase in performance. The reality, however, is not as rosy despite many attempts. Academic (and partially also industrial) tools such as LegUp [20] or Nymble [57] can synthesize subsets of their respective base languages, usually C, but lack in several aspects which makes them unviable for productive use. LegUp for example does not support access to main memory and only works on small scratchpad memories.

A lot of marketing budget is used to lure software developers into HLS by FPGA manufacturers.

In industry the situation is similarly depressing. Despite high claims of productivity increases, commercial tools such as Vivado HLS, fall short on many fronts. The FIR example from above implemented in C might look like Listing 3.4.

And the code can by synthesized just fine through VivadoHLS. However, the performance is very poor and not close to a custom accelerator. VivadoHLS can do better but needs some additional help to achieve proper performance. Listing 3.5 contains a hand optimized FIR filter with the same functionality as before. However, the hand optimized version is less of a software implementation but rather close to the hardware implementations. A shift register is used to store the state of the last iteration. Additional `pragma`'s are used to indicate to the compiler how certain loops are intended.

Both implementations can be compared by putting them through Vivado HLS 2019.1. For both versions, the target FPGA is the `xcvu9p-flga2104-2L-e` found on the VCU118 development board with a target frequency of

```c
#define N 5
void fir(int16_t *src, int16_t *dest, int elems, int16_t coeffs[N]) {
    #pragma HLS INTERFACE ap_bus port=src
    #pragma HLS INTERFACE ap_bus port=dest
    for(int i = 0; i < (elems - (N - 1)); ++i) {
        int sum = 0;
        for(int j = 0; j < N; ++j) {
            sum += src[i + j] * coeffs[j];
        }
        dest[i] = sum;
    }
}
```

Listing 3.4: Simple C implementation of a FIR filter that a user might want to synthesize into hardware through the use of a HLS tool. The code uses as few `pragma`'s as possible. Without the two `ap_bus` specifications, it would not be synthesizeable by Vivado HLS 2019.1.

```c
#define N 5
void fir(int16_t *src, int16_t *dest, int elems, int16_t coeffs[N]) {
  #pragma HLS INTERFACE ap_bus port=src
  #pragma HLS INTERFACE ap_bus port=dest
  #pragma HLS ARRAY_PARTITION variable=coeffs complete dim=1

  for(int j = 0; j < (elems - (N - 1)); ++j) {
    #pragma HLS PIPELINE II=1

    static int16_t shift_reg[N];

    #pragma HLS ARRAY_PARTITION variable=shift_reg complete dim=1
    int32_t acc = 0;
    int32_t mult;
    int16_t y;

    Shift_Accum_Loop: for (int i=N-1;i>=0;i--) {
      #pragma HLS LOOP_TRIPCOUNT min=1 max=16 avg=8
      if (i==0) {
        shift_reg[0]=src[j];
      } else {
        shift_reg[i]=shift_reg[i-1];
      }
      mult = shift_reg[i]*coeffs[i];
      acc  = acc + mult;
    }
    dest[j] = (int16_t) acc;
  }
}
```

Listing 3.5: VivadoHLS optimized FIR filter example based on https://github.com/Xilinx/HLx_Examples/blob/master/DSP/fir_example/fir.cpp. The pragmas ensure that the compiler knows how to deal with the different kinds of arrays.

Table 3.1: Comparison of the naive implementation from Listing 3.4 and Listing 3.5. Both implementations were compiled using Vivado HLS 2019.1. The optimized version can process one sample per cycle and has five cycles of delay before the result is available. The naive implementation requires 13 cycles for one sample and is not pipelined, which means it can take a new sample only every 13 cycles.

Version	Latency	Pipelined
Naive	13	No
Optimized	5	Yes

200 MHz. The results in Table 3.1 show how much of a difference the `pragma`'s and the changes to the code make. The naive implementation requires 13 clock cycles to process a single sample. This means that the implementation can process about 15 384 615 samples per second at a clock frequency of 200 MHz. The improved implementation from Listing 3.5, on the other hand, has a latency of only five cycles. However, the loop is pipelined, which means, that a new sample can be processed each cycle. Accordingly, the optimized solution can process almost 200 000 000 samples at the same clock frequency. In summary, the changes to the code and the `pragma`'s lead to a speedup of 13×.

All in all, this simple example already shows many of the problems that are encountered when targeting state-of-the-art HLS compilers. When it comes to performance requirements the base source code has to be massively changed. It is open for debate if writing hardware synthesis optimized C code or HDL directly is the easier approach.

Over the years, it became clear that it is difficult to match the performance of dedicated hardware accelerators when using generic programming languages such as C as the basis. Their target architecture is simply too different to be of any use. Domain specific languages on the other hand provide incredible opportunities by narrowing the design space and allowing for many implicit hardware specific optimizations.

3.4.2 *SystemC*

SystemC is a language, or more precisely, a library, written in C++ for hardware descriptions. It extends VHDL and Verilog by adding high level features based on the base programming language. For example, abstraction can be done through classes and templates. Furthermore, the C++ type system is available. The library adds features to support hardware constructs such as bit accurate types and interface mechanisms between different modules. Based on the library a simulation executable can be generated through standard C++ compilers. So far this sounds very much like other languages described in Section 3.3.3.

However, compared to HCLs, it is not trivial to generate Verilog or VHDL from a SystemC model. For FPGA targets, tools like VivadoHLS offer preliminary support, but this is usually even weaker than the support for general purpose C or C++. In theory the language constructs can provide additional guidance to the compiler, but in practice, the same caveats apply as they do for HLS based on general purpose programming languages.

SystemC shines when it comes to simulating large designs that are connected loosely via abstract interconnects. Since version 2 of SystemC, an abstraction of such communications based on transactions, is available that results in very fast simulation. Additionally, certain aspects of circuit design such as state machines are very easily described in SystemC. Sometimes it is not easy to distinguish how a compiler might implement a certain part of the code as the compiler has many possibilities when it comes to clocking and parallelization.

The FIR filter example can be implemented in SystemC. The main functionality of a given module is provided as a module which is a C++ class with added syntactic sugar. Specialized types are used to indicate the inputs and outputs of the module. For example, for the FIR filter we need the input data channel and the output data channel including the handshakes and clock/reset, looking like:

```
sc_in_clk    clk;
sc_in<bool>  reset;
sc_in<bool>  input_valid;
sc_in<sc_int<16>>   input;
sc_out<bool> output_valid;
sc_out<sc_int<16>>  output;
```

The module itself is created using macros in the library:

```
SC_MODULE(fir) {

  // Inputs/Outputs

  // Additional constants/variables

  // Functionality
  SC_CTOR(fir)
    {
      SC_CTHREAD(entry, CLK.pos());
      reset_signal_is(reset,true);
    }

  void entry();
};
```

The `SC_CTHREAD` macro defines a function that shall be treated as a clocked process very much like in Verilog. Additional macros exist to describe unclocked processes etc.

The functionality of the module is described in the `entry` function and looks like any other C++ function. The only addition is that the

`wait()` function is available to wait for the next clock cycle. The FIR filter can look like shown in Listing 3.6.

The language can be used to easily express complex circuits and is used for that purpose extensively by ASIC designers. The lack of tool support, however, makes the language less interesting for FPGA targets.

3.4.3 *Domain Specific Language HLS*

Designing dedicated languages for a certain domain is not a new approach and has been extensively used in many domains such as machine learning for many years. Instead of dealing with architecture details of the underlying hardware, for example a GPU, the designer has to deal with their specific problem domain only. The designers of the language have to determine which kind of features are necessary and can provide highly optimized hardware specific versions of the necessary basic building blocks. Language designers can also leave out concepts which are not necessary for the domain, such as recursion, and focus solely on what features are necessary. This behavior also makes a lot of sense for FPGA targets. DSLs designed to target FPGA can utilize highly optimized implementations of underlying primitives. For example, a DSL targeting signal processing can provide specialized implementations for filters on different number formats. The user can then use the language to specify which filters should be used in which order. The language will not be usable for other domains such as general linear algebra calculations, but will be very well suited for the specific signal processing tasks.

All in all, DSLs are very well suited to be used for high level synthesis. Accordingly, many different projects do just that. A nice feature of DSLs is that simulation and verification can usually be done right on the CPU. The necessary code can be automatically generated based on the same DSL source.

Darkroom [45] is a DSL for image processing generating hardware pipelines. Based on a simple description typical image processing algorithms such as stencil filters can be created and chained together. This approach enables the user to describe complex image pipelines with interdependencies between the stages. However, no general purpose processing is possible.

Another representative which uses an existing language is found in [92]. CellML is a language used to describe mathematical models found in biology, for example, the simulation of chemical processes inside a cell. As the language is focused on the specific domain, the authors are able to incorporate domain specific optimizations on the generated models and find representations well suited for FPGA.

In [40] a simple DSL for machine learning problems is proposed. The authors furthermore make an effort to compare the DSL approach

```
1  void fir::entry() {
2
3    sc_int<8>  sample_tmp;
4    sc_int<16> pro;
5    sc_int<16> acc;
6    sc_int<8>  shift[5];
7
8    // reset watching - Wait till execution starts
9    /* this will be an unrolled loop */
10   for (int i=0; i<4; i++)
11     shift[i] = 0;
12   result.write(0);
13   output_data_ready.write(false);
14   wait();
15
16   // main functionality
17   while(1) {
18     output_data_ready.write(false);
19     do { wait(); } while ( !(input_valid == true) );
20     sample_tmp = sample.read();
21     acc = sample_tmp*coefs[0];
22
23     for(int i=4; i>=0; i--) {
24       /* this will be an unrolled loop */
25       pro = shift[i]*coefs[i+1];
26       acc += pro;
27     };
28
29     for(int i=4; i>=0; i--) {
30       /* this will be an unrolled loop */
31       shift[i+1] = shift[i];
32     };
33
34     shift[0] = sample_tmp;
35     // write output values
36     result.write((int)acc);
37     output_data_ready.write(true);
38     wait();
39   };
40 }
```

Listing 3.6: SystemC implementation of the FIR filter. The example is taken from https://github.com/systemc/systemc-2.2.0/blob/master/examples/sysc/fir/fir.cpp and adopted. SystemC adds constructs such as waiting for a clock event to C++.

to traditional HLS tools. A simple C implementation of a vector sum function is 74 times slower than the same functionality expressed through the DSL. The HLS source can be optimized akin to the solution in Listing 3.5 which makes it 20 % faster than the DSL version. The optimized version, however, is 19 lines long compared to one line in the DSL and is not easily understandable and maintainable.

There are many more examples of DSL successfully being used and this approach has been shown to be one of the most promising directions for future research.

4 THE CASE FOR HIGHER ABSTRACTION

After reading the previous chapters the reader should have a basic understanding of the requirements of targeting FPGA. In summary changes are necessary to improve application possibilities and usability of FPGA, especially in a datacenter environment:

- Availability has to increase. Rollout as part of commodity CPUs or as cheap (and useful) add-in cards is desirable. GPUs did start out as specialized graphic accelerators for an enthusiast audience and later became the computation powerhouses we know today.
- Designing akin to assembler should not be the gold standard. Teaching modern HDLs should be focused by academia and the industry alike.
- Switch from failed HLS approaches to better optimizable and easier to use DSL approaches.
- Homogenize targeting of a variety of devices. Many applications do not need all kinds of specialized functionalities offered by modern FPGA but simply need memory access and communication facilities.
- Competition should increase through interoperability. Vendor lock-ins hurt all parties involved.
- Development of open-source toolchains at the support level of the GCC ecosystem should be focused.

A free and open ecosystem akin to GCC would increase flexibility and usability of FPGA in the datacenter.

Many of these points are known from the early software days. Nowadays, it is relatively easy to target a variety of devices using free and open software. Sharing code through platforms like Github is common and even large corporations base their business on open source software - even better - they actively fund development in open source projects to suit their own needs but also benefit from the community sharing their results on those platforms. As long as FPGA stay in their small niche with high cost of entry and limited support they will remain an outsider in the accelerator market. The vendors of FPGA toolchains are in a line with traditional silicon-focused (ASIC) CAD tools with their immensely high barriers of entry. Instead, the major players should start acting more like software companies and open up their efforts for the benefit of many. The relatively small FPGA manufacturers struggle to keep up with the demands for new and

improved toolchains whenever a new FPGA design arrives. Apart from software they also have to maintain their array of custom IP cores for a variety of applications. If large companies such as Google and Facebook can work together on complex software projects such as the Linux kernel, why are hardware companies unable to do the same?

The following subchapters focus on a different aspects of these issues faced in todays FPGA world. It shall make a case for utilizing either higher abstraction level HDLs, or directly use DSLs, instead of forcing unsuitable languages into a hardware shape.

4.1 CHANGE IN PERCEPTION

Described previously, low level HDLs such as Verilog or VHDL are comparable to the low level assembly languages of the software world. Accordingly, they pose much of the same challenges when it comes to large software projects. Excluding some exceptions in VHDL, there are no semantics to the digital circuitry described. The language can not distinguish between an bus storing an address or a floating point number. All of the logic is exclusively provided by the developer and the compiler is not capable of finding any common bugs introduced such as arithmetics with numbers of different types (e.g. signed vs unsigned). Additionally, there is no concept of interfaces. Some interfaces such as AXI can contain hundreds of signals that have to be interpreted in a specific way. In those legacy HDL the developer has to ensure all the requirements on their own. Additionally, debugging is tedious and slow as simulation runs at a very low level of abstraction and debugging tools are rather limited.

HCLs are a step away from this limited and low level approach but they face a lot of backlash in the community for a variety of reasons. First and foremost there are many developers who think that the compiler can not do a proper job of generating efficient code. These issues are very comparable to the issues faced by higher level software languages, even those that would be considered low level by todays standards such as C. However, at the software side it is known that compilers often do a much better job understanding the architecture of the processor than a typical programmer, which is necessary to write well performing assembler. Often traditional FPGA developers also fear a loss of control when they do not use low level languages. However, this notion can not hold for two reasons. Firstly, most HCLs offer the opportunity to interface with HDLs directly in the language akin to inline assembler in the software world. Secondly, most FPGA developers already use techniques such as inference which gives some control to the synthesis tool. Those techniques are relatively inefficient and coarse grained. A compiler with better semantic knowledge can optimize the code even better.

Additionally, HCLs face problems when it comes to their abstraction level. They are able to provide error checking by introducing semantics to variables as well as methods for abstracting and connecting interfaces. However, the basic concepts during development compared to HDL stay largely the same. One notable exception is Bluespec. In Bluespec the concept of guarded atomic actions offers a different way of thinking about synchronous circuits and helps reduce development time and increase productivity. However, a major problem with Bluespec is its closed source nature. The concept of guarded atomic actions is furthermore patented and can not be used by other languages. Accordingly, Bluespec has a very good product but its usage is limited to few companies and fewer universities. As the compiler is maintained by only one company, further development of the language is slow.

Closed source approaches hinder evolving the language side of FPGA targets.

Accordingly, an ideal HCL is completely open source, uses high level concepts such as guarded atomic actions, and incorporates lessons learned in software language development. Access to low level functionality should always be possible but there should be certain safety enforcements. A good example in the software world for very strict compiler guided safety practices is Rust. In Rust, certain functionalities such as direct memory access are possible, but guarded and explicitly marked by the developer. For instance, this feature might wrap unknown code written in an different HDL in a way that it is safe to use in the novel HCL. As an example, this might include certain safety guarantees regarding handshake semantics.

This new open source language should not be used by developers directly. Instead, it can be used as the basis to a bouquet of DSLs akin to the languages available for machine learning on CPUs and GPUs. The novel language should be used to write highly optimized primitives that are employed in the DSLs by any interested party. Using the DSLs should be easy and without a high need for hardware knowledge.

Lastly, open source collaborations such as open hardware should be intensified. A big issue when employing IP from different parties are the lack of common interfaces. Simply moving from one IP to another often involves many changes to the underlying interface logic. Accordingly, these projects should focus more on enforcing selected interfaces. For example there can be a group of AES encryption cores provided by a variety of developers. Currently, each core might use different interfaces and different configuration methods. The platform should then enforce one standardized interface for all the cores to make them easily interchangeable and comparable. This process would help not only FPGA developers but also FPGA research. At the moment it is often impossible to reproduce certain results of FPGA papers as the development environment is very specific. Afterwards, FPGA targeted conferences could enforce that cores for a certain application adhere to the standardized interface. Akin to projects such as the Middleburry Stereo Evaluation [105], the cores could be automatically verified and

Switching from one implementation of an accelerator to another one is difficult as there are often no standardized interfaces.

benchmarked in online comparison websites to increase trust in the published results.

Most of these changes target those educated in FPGA design. However, the far wider audience is not knowledgeable about FPGA. Consequentially, it would be very detrimental for the overall adoption of FPGA to ignore such a large audience. The next section focuses on the particular needs of such an audience and refines previously made points about DSLs.

4.2 FPGAS FOR NON-FPGA EXPERTS

The big two FPGA vendors are certainly aware of the adoption problems faced when scaling up FPGA deployments. Their current take on the issue is high level synthesis tools that take languages used to target CPUs or GPUs, such as OpenCL, and mangle them to target FPGAs. However, the performance achieved by those languages can be poor [102]. Additionally, the code that ends up going into the compiler looks nothing like a software implementation would (See Chapter 3). Accordingly, the soon to be FPGA programmer has to learn the specific dialect of the language that works well when compiled onto the FPGA. Besides, this approach is not new and research in this area is almost as old as FPGA themselves [42]. In spite of the huge research efforts involved, no major breakthroughs have been achieved. Summarized, programming language based HLS comes down to:

- Pro:
  - Languages are well known.
- Contra:
  - Languages are unsuited to target FPGA, the programmer has to learn a specific dialect and know hardware details. The time this takes could be spent on directly learning HDLs.
  - Even highly specialized code written for HLS often performs poorly compared to hand-crafted HDL IPs.

In brief, not much is won using HLS over HDL.

Instead of misusing software languages in HLS, specialized DSLs should be used by non FPGA experts.

Instead the community should learn from other fields where specialized accelerators are heavily used. The prime example of such a field is machine learning. GPUs are the primary accelerator utilized, but a plethora of other, dedicated accelerators are also available [91]. From a programmers point of view nothing changes when using a different accelerator type apart from the execution performance. The code is not written in some general purpose language but instead in highly specialized DSLs that support exactly the features needed for the specific domain. This approach benefits hardware designers as they can focus on writing highly optimized cores for certain important

tasks and use a compiler to stitch these cores together for the given application. This approach already produces promising results [115].

One aspect of DSLs design for FPGA targets that should be considered quite well is the abstraction of the device. Nothing is won when the DSL targets only one specific FPGA and does not scale across a variety of devices. Accordingly, there should be some intermediate representation that is independent of the FPGA fabric and interfaces with an abstraction layer. Chapter 5 presents such a tool, TaPaSCo, and explains how abstraction of different FPGA architectures can look like.

The rest of this chapter presents a typical development process targetting FPGA. The case study shows that seemingly trivial tasks, such as moving data around between two different memories, is very error prone on FPGA. Hence, tools such as TaPaSCo should be used to benefit from the efforts of other people instead of reinventing the wheel another time.

4.3 CASE STUDY: NOTHING IS STRAIGHT FORWARD ON FPGA

An FPGA on its own without any connection to the outside world is rather pointless. Accordingly, moving data in and out of the FPGA is a fundamental operation. For the PCIe based devices, TaPaSCo (See Chapter 5) utilizes a Direct Memory Access (DMA) unit to move data from host memory directly to device memory. A schematic of its operation is presented in Figure 4.1 The DMA engine is located in the device between the PCIe interface and the DDR memory. A transfer is done by initializing the DMA unit with the host and FPGA memory addresses, the transfer direction and size. The ensuing process can be as straight forward as: Start the transfers on the respective interfaces and move the data to the target location.

However, the real world is not as simple. Nowadays Linux does not give access to physically continuous memory to the user space. Instead, a user virtual memory is assigned that is usually not physically continuous, and not easily accessible by the PCIe attached FPGA. Even worse: The kernel does not like to allocate physically continuous memory, as it is a very scarce resource. Accordingly, 4 MB is a typical maximum size of those memory regions. So how does it work if more than 4 MB need to be transfered? Two frequently used techniques are bounce buffering and scatter-/gather-lists. For the latter a list of the pages in user space memory can be directly transferred to the FPGA and the device has to make sure that the correct pages are transferred (Illustrated in Figure 4.2). The advantage is that no copies between host memory locations take place but the DMA unit has to be more complex. TaPaSCo goes with the other approach and utilizes a bounce buffer. For this approach the user memory is moved *in chunks* to the

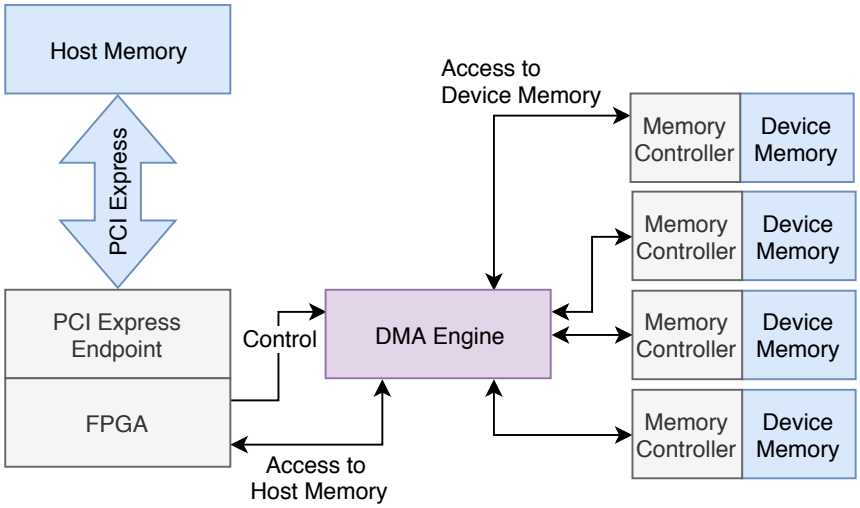


Figure 4.1: Overview of the DMA system on the FPGA. The host memory is accessible via PCIe 3.0. The DDR memory on the device is accessible via memory controllers. The DMA engine moves data from one memory to another. Blue boxes are not on the FPGA. Gray boxes are provided by the vendor, in this case Xilinx. The operation is controlled by the host over the PCIe link.

continuous memory, and then by the DMA engine to FPGA memory as shown in Figure 4.3.

TPC uses a simple DMA implementation called DualDMA that is able to handle a single transfer and has no queue to store multiple requests. TaPaSCo introduces BlueDMA as a replacement with additional features and improved performance. The following case study focuses on exploring various design decisions and makes a case for not reinventing the wheel as achieving optimal performance on an FPGA is not as trivial as it sounds even in less complex scenarios.

Whereas PCIe 3.0 x8 promises 8 GB/s of bandwidth, the eventual bandwidth shrinks to 7.306 GB/s due to encoding and packetization overhead [21].

Going back to bounce buffering. A design to implement a DMA engine using bounce buffering could work accordingly: (1) Allocate a large chunk (typically 4 MB) of continuous memory, (2) move chunk from user space memory to continuous memory, (3) tell DMA engine to move chunk to correct location on FPGA memory, (4) repeat (2) to (3) till all chunks have been transferred. This approach certainly works and achieves around 17.8 % and 17.3 % for reads and writes respectively of the practical PCIe 3.0 x8 bandwidth of 7306 MB/s [21] as plotted in Figure 4.4. This peak value is reached at a transfer size of only 256 kB and drops off considerably for larger transfers and for smaller transfers as well.

Accordingly, the performance is not even close to what can be expected of PCIe. Finding out the reason for limited performance is often one of the most difficult tasks when targeting FPGA. The first debugging step usually focuses on simulation: Does the DMA

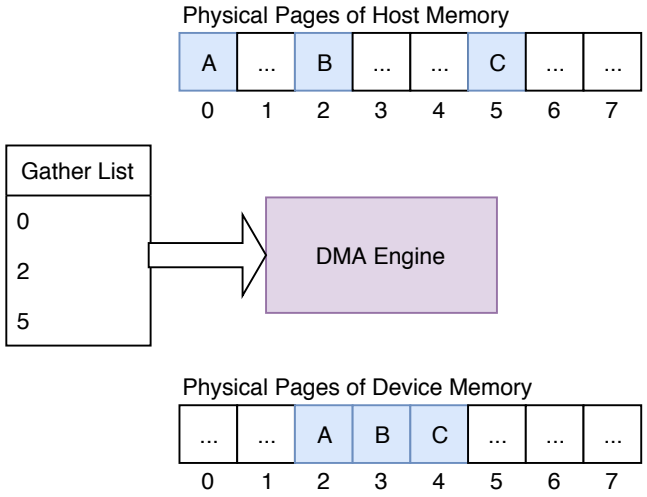


Figure 4.2: Example of using a gather list to move data from user space of the host to the device memory. The pages corresponding to the continuous user space memory are not continuous on the physical memory. In this case, the pages A, C and F which represent a continuous buffer in user space, shall be transferred to a continuous block starting at address two in device memory. The DMA engine receives a list of memory addresses that have to be transferred and automatically moves the pages to the corresponding addresses in device memory.

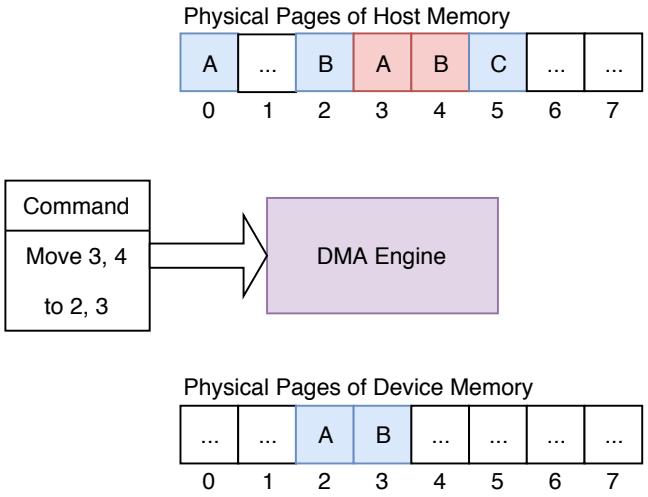


Figure 4.3: Example of using a bounce buffer to move data from user space of the host to the device memory. The bounce buffer, represented by the pages marked red, can hold two elements. Hence, the kernel driver has to move the data in two chunks from the user space to the continuous buffer. In this case, the first chunk consists of the pages A and B. The DMA engine then moves the data from the continuous buffer to the device memory. This process has to be repeated until all chunks have been transferred.

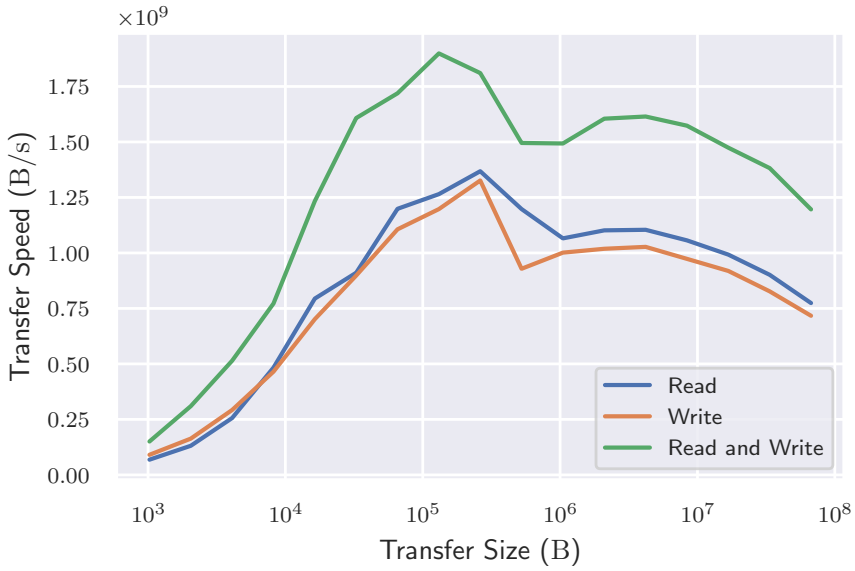


Figure 4.4: Performance for Read, Write and both in parallel over PCIe 3.0 on a AMD Athlon(tm) X4 845 for a bounce buffer of 4 MB. The measurements include the whole process of moving data from user space to the FPGA and/or back.

engine behave strangely in simulation? Can a scenario be created that reproduces the behavior of the device? Simulating the complete system is not trivial as it requires accurate models of all parts involved. For the design under user control the required models are immediately available. However, simulating the parts that are provided by the FPGA vendor is only possible at the mercy of the IPs creators. The PCIe interface comes with a simple functional model which can not be used as it is not accurate enough to find performance problems experienced during tests with the device. Even less information is available about the rest of the chain leading up the host such as bridges and root complexes. Accordingly, in this case simulation can only find flaws in the user logic part, but not in the whole system. The tests showed that the DMA engine performs in the expected range and was capable of exceeding the performance required for PCIe 3.0 x8.

As simulation was not able to resolve this issue in this case, the slow and error prone process of in hardware debugging followed. Looking into the FPGA during processing is possible by utilizing vendor provided logic analyzers that can be introduced into the bitstream during generation. These logic analyzers can capture some specified signals during execution. Sampling starts based on specified triggers such as signal changes or combination of signals. The major downside of the logic analyzers is that they often increase bitstream generation times. Additionally, one logic analyzer can only cover a limited number of signals and will not be able to capture all signals of interest at the same time, leading to multiple bitstreams for a thorough investigation.

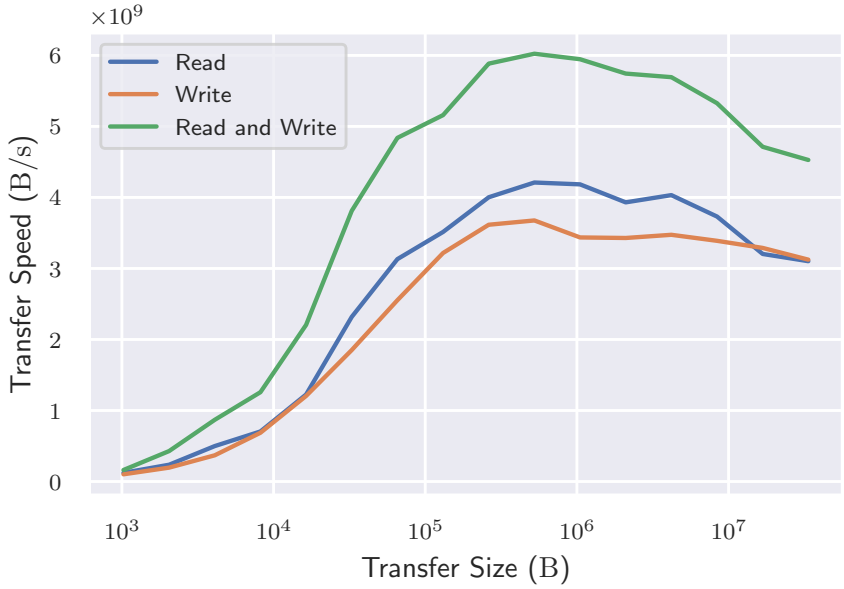


Figure 4.5: Performance for Read, Write and both in parallel over PCIe 3.0 on a AMD Ryzen 1600x for a bounce buffer of 4 MB. The measurements include the whole process of moving data from user space to the FPGA and/or back.

Accordingly, in hardware debugging should be considered as a last resort after other debugging methods did not find the issue.

In hardware debugging indeed found one issue: During reads, the PCIe bridge provided data only on every *other* clock cycle. What followed was a deep dive into the technical documentation of all parts involved. In the end it turned out that the host platform is not capable of sustaining the required data rates. Moving to a newer platform led to the performance shown in Figure 4.5. However, the graph, which was generated after moving to a better performing platform, also shows that this was not the only issue leading to poor performance.

Digging deeper brings another issue to light. The transfers themselves are pretty fast and frequently sustain the expected PCIe 3.0 x8 bandwidth. However, the host is not able to keep up with the demand and the time between two transfers rapidly adds up. This issue is already known from [21] and was previously resolved by using multiple independent DMA engines. This approach however is wasteful, as each DMA unit is already capable of achieving the required bandwidth. Any other DMA engine has to wait in line till the previous one has finished, basically not exploiting any parallelism.

Instead, for BlueDMA another approach is taken. The engine has a *command queue* that stores requests which can be processed as fast as possible one after another. This addition requires changes to the driver in the host system as well. The original bounce buffering steps have to be amended. Instead of allocating only one large chunk of 4 MB,

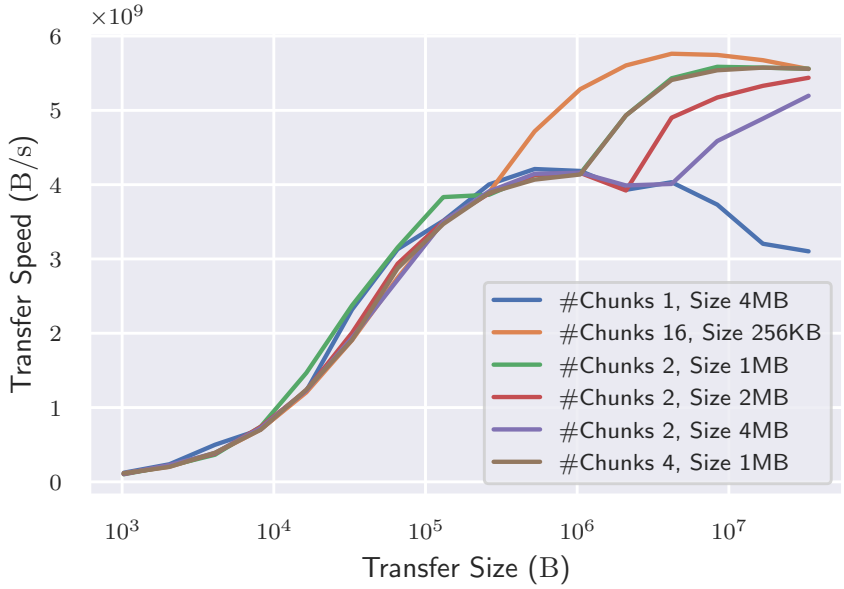


Figure 4.6: Performance for Reads (Moving data from the FPGA to the host) over PCIe 3.0 on a AMD Ryzen 1600x for different configurations of bounce buffers. The measurements include the whole process of moving data from user space to the FPGA and/or back.

multiple chunks of various sizes are allocated. The driver automatically selects the best size for a host system based upon previous benchmarks. The host can then fill the next buffer and queue it in the FPGA while the DMA engine is busy servicing the requests.

Figure 4.6 shows transfer speed achieved for different buffer sizes for reads. Writes and parallel Reads and Writes improved on the same line and were not plotted for better readability. Interestingly the performance is much better when using many small buffers instead of few large buffers as the load is better distributed between the involved parties. The difference between the optimal bandwidth and the achieved bandwidth shrinks to 78.9 % and 65.4 % for reads and writes respectively. For the best performing configuration of 16 chunks of 256 kB the transfer speeds degrade much less for larger transfers and stay approximately linear for transfers up to multiple GB. There is still a rather noticeable difference between promised and achieved speeds, especially on the write side, requiring further investigation. Adding additional techniques to reduce the host overhead such as zero-copy buffers instead of bounce buffers can further increase the bandwidth utilization [21].

The weaker system shows a similar increase in performance using multiple small chunks compared to a single large chunk. However, the weaker CPU does require a different setup and number of bounce buffers. Accordingly, platform specific driver initialization has to be

employed and testing on a variety of platforms has to be done to find optimal configurations.

This case study highlights certain aspects of FPGA design. Even tasks, that appear simple at first, such as moving memory around, have a rats-tail of design decisions which heavily influence the performance and are often difficult to track down. Hence, instead of reinventing the wheel, FPGA developers should work together to build well working infrastructure that can be reused for different purposes, evolving from its closed source IP driven past to a collaborating future for the benefit of all researchers.

5 TASK PARALLEL SYSTEMS COMPOSER (TAPASCO)

Parts of this chapter have been published in:

1. Jens Korinth, Jaco Hofmann, Carsten Heinz, and Andreas Koch. "The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems." In: *International Symposium on Applied Reconfigurable Computing (ARC)*. 2019

Targeting an FPGA requires more work than just implementing the desired functionality as an IP core and hitting the synthesis button. Without the required on-chip infrastructure to communicate with the FPGA, it is impossible to control the execution of the IP core. Additionally, access to off-chip memory or other peripheral devices is not available by default. Hence, the user needs some kind of support layer that provides all the surrounding logic around the IP core itself. Traditionally, the peripheral design is hand crafted for that one IP core on one device. Accordingly, the design is hard to port to a different FPGA as this step requires the complete change of the surrounding infrastructure.

Commercial tools that try to make IP portable, such as SDAccel or SDSoC, exist, but they have a narrow scope. The former supports only a small number of PCIe-based boards, while the latter works only on Zynq based devices.

The project presented in this chapter, TaPaSCo, provides a framework that can be used to easily build portable and powerful FPGA designs. An IP core that is compatible to TaPaSCo can be run on a many different platforms based on Zynq, MPSoC or PCIe without any changes.

5.1 CONCEPT

The concept behind TaPaSCo, the Task Parallel System Composer, is the automated generation of System-on-Chips (SoCs) based on individual accelerators, called Processing Elements (PEs), for compute applications. Each design consists of a number of PEs of different types. The type and the number of PEs per type can be freely configured.

A TaPaSCo compatible PE is shown in Figure 5.1. Three interfaces are provided for communication with peripherals or the host.

Firstly, the host has to be able to control the PE through a control channel. On all current platforms, this channel is realized as an AXI4-Lite interface. Secondly, the PE can access off-chip memory to

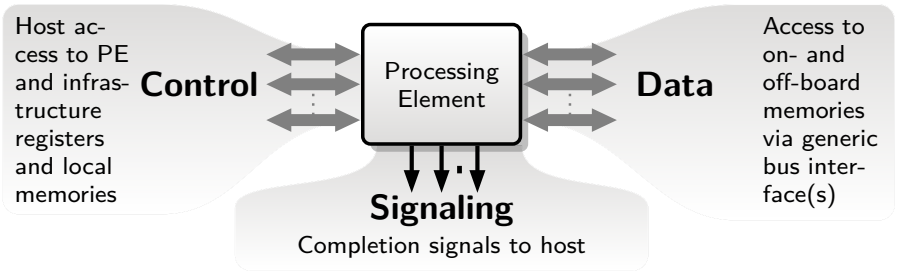


Figure 5.1: A TaPaSCo compatible PE has to conform to this T-shape design. The PE is controlled via control channels that can be accessed by the host. The PE itself can access off-chip memory through the data channel. Lastly, signaling paths can be used to notify the host. Taken from [69].

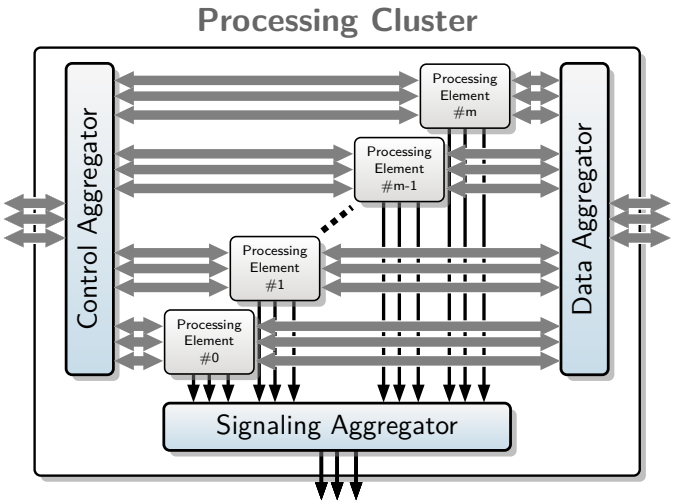


Figure 5.2: TaPaSCo PEs are grouped together in a processing cluster. The individual channels are aggregated into a combined channel that can be connected to the rest of the architecture. Taken from [69].

access input parameters or store results. This channel is currently implemented as AXI4-Full. Lastly, a signaling channel is used to notify the host when a PE has finished execution. This channel uses a platform specific interrupt mechanism, for example MSI-X for PCIe based devices.

This shape fits many accelerators already, but some accelerators might need a wrapper to be compatible. If an accelerator uses streaming instead of memory mapped data accesses, it can be connected through a wrapper that transforms the memory mapped access into a stream, as done in Chapter 6.

Processing elements are combined to form a processing cluster. As illustrated in Figure 5.2, the overall shape is comparable to the PE shape. However, the individual data, control and signaling channels are aggregated to form a single channel.

So far, the PEs are not connected to the outside world. The TaPaSCo platform contains the necessary infrastructure to connect the PEs to

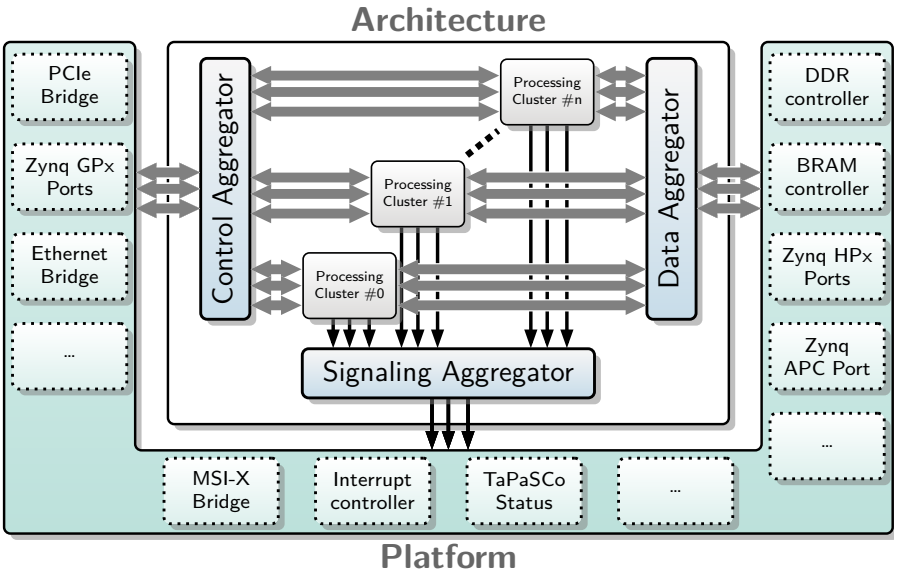


Figure 5.3: TaPaSCo design containing the architecture which houses the processing clusters, as well as the platform. The platform is FPGA specific and includes different components depending on the target architecture. In any case, it handles all communication of the PEs with the outside world. Taken from [69].

the host and external memories. The details of the platform depend on the target FPGA and can look very differently to one another.

An abstract view of the platform is shown in Figure 5.3. The individual implementation is device specific. For example, a PCIe based board such as the Alveo U250 utilizes host access through PCIe. External memory is attached as DDR4 memory through specialized memory controllers. As the FPGA does not share memory with the FPGA, data that needs to be accessible by the PEs, has to be moved from host memory over PCIe to the device memory.

A Zynq based device, on the other hand, is tightly integrated. The ARM cores that act as host are directly connected via AXI connections with the PEs. Memory access is also handled through provided AXI connections, and no specialized controllers have to be instantiated.

As TaPaSCo assumes nothing apart from the T-shape about PEs, the framework can be used for a wide number of applications. There is also no limit on how few or many PEs are used. It is even feasible to use only one PE containing a complete system.

Accordingly, PE can look very differently. For instance, a high performance stereo vision accelerator based on systolic arrays as shown in Chapter 6 is a classical stream based accelerator with a TaPaSCo compatibility wrapper. Another application are RISC-V based soft core processors which are wrapped as TaPaSCo PEs [69].

TaPaSCo attaches to the vendor provided tools to build the bitstream. Currently, all supported platforms are based on Xilinx FPGA and TaPaSCo uses Xilinx Vivado. The bitstream generation process itself

is very flexible and offers multiple points to attach plugins. One example is presented in Section 5.7, where the plugin system is used to bring Ethernet support to TaPaSCo. Other typical examples are device specific routing optimizations on the VC709 platform, or the inclusion of an OLED display on the ZedBoard.

5.2 SOFTWARE

Generating a bitstream for a platform through TaPaSCo requires the execution of only one command:

```
1 tapasco compose '[counter x 2, arrayinit x 2]' @ 350MHz -p AU250
```

The command generates a bitstream running at 350 MHz for the Alveo U250 platform. The bitstream contains two `counter` PEs and two `arrayinit` PEs.

TaPaSCo needs to know about PEs before usage, by running the `import` command. This command will make sure that TaPaSCo knows the necessary details about the PE.

But what happens if the designer does not know what the maximum frequency of the design is or how many PEs the target device fits? For this purpose, TaPaSCo provides a design space exploration (DSE) feature [69]. Depending on the requested exploration dimensions, TaPaSCo finds suitable PE configurations for the target device. Successful applications of this feature are presented in Chapter 6 or in different papers, for instance in [46, 115].

5.3 FROM TPC TO TAPASCO

TaPaSCo evolves from TPC by supporting many different FPGA under a common framework. IP developed once, can be used on any current or future platform.

TaPaSCo started in 2013 as Threadpool Composer (TPC) as part of the Repara project [68] funded by the European Unions Seventh Framework Program (FP7). The project's goal was to homogenize targeting of a variety of accelerators such as DSPs, GPUs and FPGAs. TPC provided a framework abstracting different FPGA, namely, one PCIe based platform, the VC709 development board with a Virtex 7 FPGA, and two Xilinx Zynq based platforms, the ZC706 and ZedBoard. The three boards can be targeted homogeneously through a common software and hardware Application Programming Interface (API). The user provides functionality in the form of PEs. The PEs follow a specified hardware interface, and everything else, such as the infrastructure to talk to a host, is taken care of by TPC. A PE once integrated in TPC can be used on any of the supported FPGA without any additional changes.

TaPaSCo abstracts different FPGAs during bitstream generation and software development.

The software is also abstracted: Apart from recompilation of the library and user space programs, no additional changes are necessary, if the host architecture changes. With the completion of Repara, TPC was rebranded to TaPaSCo and development focused on improving

TPC. As the project originally targeted only two types of devices, the code was in many parts specific to either of the platforms. For example, there had to be two drivers, one for the ARM based Zynq host and one for the x86 based PCIe host. Accordingly, the software and drivers had poor modularity. Additionally, the user space library required knowledge about the platform it was running on, leading to separate code paths for the different platforms which resulted in poor maintainability. Only the Zynq platform had minimal abstraction as two instances, the ZC706 and the Zed-Board, of it existed. However, the PCIe path consisted of only one member and adding an additional PCIe based device was tedious.

The first problem tackled is the addition of more platforms under the existing umbrella, as well as supporting entirely new generations of FPGA. Besides Zynq and PCIe, TaPaSCo supports MPSoC, which is an improved version of the legacy Zynq devices. MPSoC replaces the two Cortex-A9 cores with four Cortex-A53 and two additional Cortex-R5 for real-time work. The fabric has been upgraded from seven series to UltraScale+.

TaPaSCo introduces a base PCIe platform that simplifies adding new platforms. Nowadays, the addition of a new PCIe based platform requires the specification of only a few key parameters such as memory configuration and PCIe core settings and everything else is derived automatically.

The second major change, which will be introduced in more detail here, improves upon the user space and kernel software. As previously mentioned there is a lot of duplicated code in the legacy TPC versions of the software. TaPaSCo introduces a unified software suit that thoroughly abstracts different devices and provides an abstracted interface to the user. The current version of TaPaSCo shares most of the code paths for all the different device types. The TaPaSCo Loadable Kernel Module (TLKM) automatically determines which platform it is running on and extracts the necessary information about the bitstream and the FPGA directly from the bitstream. Adding a new architecture is as easy as adding the address of the information storage to the bitstream. The complete software stack is shown in Figure 5.4.

5.4 BITSTREAM IDENTIFICATION

In contrast to ASICs one FPGA can house very different applications. The kernel driver can dynamically extract necessary information directly from the device.

Initial versions of the project required very strict adherence to certain address specifications. For example, there was support for four DMA engines at specified places in the device's memory space, all the PEs had to be placed in a certain order and had only 4 kB of configuration memory each and so on. Accordingly, changing anything about the structure of the addressmap required touching of many different code locations. As a first step towards a flexible addressmap in TaPaSCo a status core was introduced that was supposed to contain the necessary

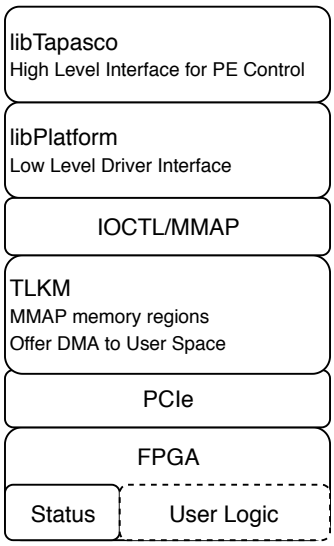


Figure 5.4: View of the TaPaSCo software stack. The bitstream generated by TaPaSCo contains a dedicated information core providing all necessary information to the host at run-time. The kernel driver reads that information and provides necessary interfaces, such as interrupts or DMA, to the user space library. The user space library provides high level operations to control the FPGA to the user application.

addresses and provide them to the host. The host could then initialize the devices based on the information of the status core. Accordingly, the host did not need to know the addresses of any other component and could derive any other information based on the address of the status core. The initial version of the status core was using a custom binary representation of the information and was already able to improve the flexibility of the software and hardware stacks. However, the custom format soon turned out to be too restricted and adding information required many changes to all parts of the software. Furthermore, the custom IP that provided the information, written in Scala, turned out to be somewhat unstable and difficult to use.

Based on this experience a second version of the status core has replaced the old one. Instead of a custom IP, all of the information is located in a simple BRAM ROM which can be read via AXI by the host. Furthermore, instead of using a custom format, a highly specified and popular format for data serialization is used, namely Protobuf [58]. Even though there are other formats that promise higher performance and lower size, Protobuf was selected because of very light weight libraries that are available, which are suitable for usage in the kernel space. As TaPaSCo needs to read the information in kernel space as well, this is one important requirement.

TaPaSCo has to extract the necessary information about the bitstream duing the bitstream generation process. However, as the vendor tools used are controlled through the TCL programming language,

```
{
  "Timestamp"           : 1566432000,
  "InterruptControllers" : 1,
  "Versions"            : [{
    "Software"     : "Vivado",
    "Year"         : 2018,
    "Release"      : 3,
    "ExtraVersion" : ""
  },{
    "Software"     : "TaPaSCo",
    "Year"         : 2019,
    "Release"      : 6,
    "ExtraVersion" : ""
  }],
  "Clocks"              : [{
    "Domain"    : "Host",
    "Frequency" : 250
  },{
    "Domain"    : "Design",
    "Frequency" : 100
  },{
    "Domain"    : "Memory",
    "Frequency" : 300
  }],
  "Capabilities"        : {
    "Capabilities 0" : 12
  },
  "Architecture"        : {
    "Base"        : 0x0000000002000000,
    "Composition" : [{
      "Type"   : "Kernel",
      "SlotId" : 0,
      "Kernel" : 14,
      "Offset" : 0x0000000000000000,
      "Size"   : 0x0000000000010000
    }]
  },
  "Platform"            : {
    "Base"       : 0x0000000000000000,
    "Components" : [{
      "Name"   : "PLATFORM_COMPONENT_STATUS",
      "Offset" : 0x0000000000000000,
      "Size"   : 0x0000000000002000
    },{
      "Name"   : "PLATFORM_COMPONENT_INTC0",
      "Offset" : 0x0000000000020000,
      "Size"   : 0x0000000000010000
    },{
      "Name"   : "PLATFORM_COMPONENT_DMA0",
      "Offset" : 0x0000000000010000,
      "Size"   : 0x0000000000001000
    }]
  }
}
```

Figure 5.5: JSON configuration of a bitstream generated through TaPaSCo. The JSON-file is then serialized using Protobuf into a binary representation that can be read from the bitstream by the software stack.

and there are no TCL libraries for Protobuf, an intermediate step is necessary. After the infrastructure generation is completed, a JSON file is generated which contains all information about the current bitstream needed for software execution. The JSON-file, an example is shown in Figure 5.5, is then serialized using a custom program that parses the JSON, serializes it to Protobuf and writes it to a file that can be read again by the vendor tool flow to initialize a BRAM. The resulting binary representation is often smaller than the original custom binary format while being more flexible.

The configuration can be extended by the user through a plugin system.

The bitstream in the example JSON-file in Figure 5.5 contains one PE with the ID 14 at address 0x2000000. Furthermore, the bitstream contains three platform components: (1) the status core itself. Although

the address has to be known beforehand to read the status information, the address is kept for completeness, (2) an interrupt controller that is required by the TaPaSCo hardware API to signal the host completion of PE execution, and (3) a DMA controller that can be used to move data from host to and from FPGA memory. Moreover, some information about the generation process itself is added (versions used, bitstream generation time, clock frequencies). Lastly, a capabilities field is kept for compatibility reasons to older TaPaSCo versions. The number 12 in this case refers to the bitstream using local memories and having a dynamic address map, in contrast to the static address map used in older versions.

The JSON, and the derived binary, can easily be extended to contain additional platform components through a plugin system to inject user provided platform components into the bitstream generation. Custom components are used to transparently provide additional features, such as Ethernet controllers. The relevant information to use those components in a user application are provided further down the stack through an API described hereafter.

5.5 ABSTRACT KERNEL DRIVER

As mentioned at the start of Chapter 5 the drivers of TPC are not easy to maintain and contain a lot of duplicated code. Accordingly, TaPaSCo replaces the different platform drivers with a new driver that works for all the different platforms. To do so the driver has to determine what kind of platform it is running on. This is done through either PCIe IDs for PCIe based devices or a special device tree node for Zynq and MPSoC based devices. Depending on the platform a special initialization sequence takes place. For example for PCIe the MSIx interrupts are initialized. The rest of the initialization sequence is the same for all platforms. The driver contains the address of the status core for a given platform. This address is used to read the protobuf information from the status core which is then parsed in the driver and used for the rest of the initialization. First of all, the memory segments that contain the platform components and the architecture are mapped into kernel space using `mmap`. Furthermore, special components such as DMA engines and interrupt controllers are initialized based on the status core information. The driver is deliberately kept thin to avoid unnecessary latency, only the functionality that can not easily be moved to the user space remains in the driver.

The driver provides the interfaces shown in Table 5.1. The first type `mmap` maps the FPGA memory segments into user kernel space, as it was done with kernel memory space before. The second type provides functionality through `IOCTL` which are basically special function calls that are called from user space and are executed inside the kernel and might have parameters as well as return types.

Name	Type	Function
VERSION	IOCTL	Print TLKM version information
ENUM_DEVICES	IOCTL	Print available FPGA
CREATE_DEVICE	IOCTL	Acquire FPGA
DESTROY_DEVICE	IOCTL	Release FPGA
INFO	IOCTL	Print acquired device info
SIZE	IOCTL	Retrieve sizes of memory regions
COPYTO	IOCTL	Copy from user space to device
COPYFROM	IOCTL	Copy from device to user space
DEVICE	MMAP	Map control memory regions
USER	READ	Retrieve information about PEs execution status

Table 5.1: Interfaces offered by the driver to the user space. The functionality is very low level and should not be used directly by a user. Instead `libtapasco` is provided as an intermediate layer talking to the driver and providing high level functionality.

5.6 USERSPACE LIBRARY

The user-facing side of the TaPaSCo runtime consists of `libtapasco`, a C and C++ library providing high level functionality to user applications. Furthermore, there is a second in-between layer in `libplatform` that can be used to access lower level functionality such as reading and writing device registers. The libraries follow the goal to avoid any connections between the usage of the library and the target device. All hardware details are supposed to be abstracted. Accordingly, device addresses are not directly used. Instead, generic addresses, which remain the same for all platforms, are translated to device addresses by translation functions. Using this method, the library does not need to be aware of the hardware details and can be reused on any of the supported platforms.

However, the goal is that the user does not have to deal with any addresses at all. Accordingly, the high level API works in a different way. TaPaSCo PEs are abstracted as C++ function calls. The user calls a special TaPaSCo-launch function that has one or two set parameters and an additional variadic argument list for any other parameter the function might need (see Figure 5.6). Parameters come in two types right now: values or arrays. Value parameters can be directly used as arguments to the function and will be written into the corresponding configuration registers of the PE. The second type, arrays, require additional work. They consist of a pointer to a block of memory and the size of the array. TaPaSCo then takes the user space pointer and makes sure the data is available to the FPGA and writes the translated,

Segment	Purpose	Address User Space	Address PCIe	Address Zynq
Platform	Contains TaPaSCo infrastructure	0x10000000	0x0	0x80000000
Architecture	User-Provided Logic	0x8000000	0x2000000	0x40000000

Table 5.2: Addresses used on the the PCIe and Zynq TaPaSCo platforms and their corresponding user space addresses. The addresses of the different platforms have to address the specific details of the host to FPGA connection, for example AXI, while the user space address is unified for all platforms.

```
1 template <typename R, typename... Targs>
2 job_future launch(tapasco_kernel_id_t const k_id, RetVal<R>
  ↪ &ret, Targs... args);
3 template <typename... Targs>
4   job_future launch(tapasco_kernel_id_t const k_id, Targs...
    ↪ args)
```

Figure 5.6: TaPaSCo launch functions to run a PEs on the FPGA.

FPGA accessible, address into the corresponding configuration register of the PE. Translation is done using the features of the specific platform. For shared memory devices such as Zynq, this process uses kernel provided DMA addresses and does not include memory movement. However, for PCIe based devices the process requires bounce buffered memory copy with the corresponding overhead. As it is often not necessary to move the same array to the device and back, there are additional specifiers that let the user specify an array as input only, output only or bidirectional. The data is moved using bounce buffering as shown in Figure 4.3.

The previous sections described one possible way of abstracting away the software side of targeting FPGA. Instead of letting the user deal with the kind of device at hand, the information is retrieved directly from the hardware. Of course this is not the only possible solution for this problem. For example, GPU vendors also have a lot of largely similar devices over different families that have to be supported by their drivers and have to look the same through the software APIs. However, as GPUs are not re-configurable, the relevant information is often put into the driver instead of inside the chip as the functionality will not change for the same device.

5.7 THE TAPASCO PLUGIN SYSTEM: SFP+

The plugin system of TaPaSCo can be used to bring new features to any supported platform. The plugins can attach to many different stages of the bitstream generation process. For instance, plugins can be called right at the start, before synthesis, before implementation or after finalizing the bitstream. Additional attachment points are also available.

The Ethernet support, that is available on the NetFPGA SUME, ZC706 and VC709 platforms, is one notable example of the plugin system in action. The plugin improves the base platform by adding all the necessary infrastructure to attach the FPGA to SFP+ connectors. This behavior is outside the scope of normal TaPaSCo PEs as presented in Figure 5.1.

In this case, the plugin is used to attach selected PEs the the IP representing the SFP+ ports. On Xilinx FPGA, the interface is controlled by a vendor provided IP that provides one streaming interface per direction. Thus, one interface for sending and one for receiving of Ethernet traffic. Ethernet frames arrive at the accelerator as a stream of 64 bit words, in the case of 10 Gbit/s Ethernet.

A PE that is compatible with this plugin needs two additional interfaces in addition to those found on normal PEs. The PE needs two AXI4-Stream interfaces, one for each direction. The receiving interface is fed with Ethernet frames, while the sending interface can be used by the accelerator to send out its own Ethernet frames. However, the plugin also supports multiple SFP+ connections per PE to realize complex Ethernet attached PEs such as smart switches.

A sample system is illustrated in Figure 5.7. The system uses two types of PE in this case. The first type uses two Ethernet connections, for instance, the PE could perform traffic monitoring tasks. The second type of PE uses only one Ethernet connection. This PE could implement a network attached key-value store.

The configuration consists a list of individual port configurations. Each port requires the field `PORT` which specifies the physical port the PEs should be connected to. Additionally, the `mode` has to be specified. There are currently three connection modes: `singular`, which connects a single port with a single PE, `roundrobin`, which connects multiple PEs with a single port and arbitrates between them in a round-robin fashion, and lastly, `broadcast`, which forwards all incoming packets to all PEs.

The next configuration option is `sync` which specifies in which clock domain the PEs should run in. If `sync` is true, the PEs is clocked with the PE clock and synchronization between the SFP+ domain and the PE domain is done in an instantiated conversion circuit. The other option is, that the PE receives three clocks and is responsible for

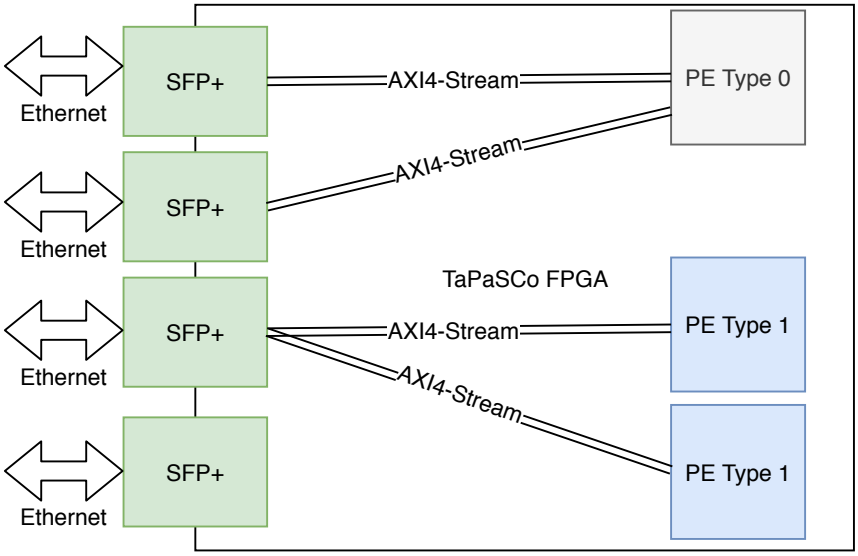


Figure 5.7: Ethernet is attached through special IP provided by the FPGA vendor, here marked in green. The SFP+ IP provides two AXI4-Stream interfaces for bidirectional communication. The TaPaSCo plugin can flexibly connect different types of PEs to the ports. The user can specify which ports should be connected to which PE.

synchronization itself. The three clocks in this case are the PE clock, the receive clock and the transmit clock.

Lastly, the list of PE that should be connected has to be specified as `kernel`. Each PE entry consists of the PE ID, the number of PE that should be considered, and the interfaces that should be used for the receive and transmit channels.

The plugin system allows that custom plugins are fed with parameters that are provided by the user either through configuration files or the command line. In this case, this feature is used to specify which PE should be connected to which physical SFP+ port.

Furthermore, the plugin supports different modes of operation, to support different system requirements. For instance, two key-value store PEs can be attached to one SFP+ port to balance the load. For this case, the plugin provides the round-robin mode which inserts arbitration between the PEs and the port. An example configuration of this scenario is shown in Listing 5.1. The other PE is directly attached to a single SFP+ port.

The system is automatically built based on the configuration. TaPaSCo calls the different stages of the plugin during bitstream generation at the specified time. The plugin itself uses two functions to implement the desired functionality:

```
1 tapasco::register_plugin
  ↪ "platform::sfpplus::validate_sfp_ports" "pre-arch"
2 tapasco::register_plugin "platform::sfpplus::addressmap"
  ↪ "post-address-map"
```

```
{
  "SFP1": {
    "PORT" : "0",
    "mode" : "singular",
    "sync" : "true",
    "kernel" : [{
      "ID"  : "EthernetMonitor",
      "Count" : "1",
      "interface_rx" : "axis_S",
      "interface_tx" : "axis_M"
    }]
  },
  "SFP2": {
    "PORT" : "1",
    "mode" : "roundrobin",
    "sync" : "false",
    "kernel" : [{
      "ID"  : "KeyValue",
      "Count" : "2",
      "interface_rx" : "axis_S",
      "interface_tx" : "axis_M"
    }]
  }
}
```

Listing 5.1: Example configuration for three PE that are attached to SFP port 0, 1 and 2. PE one is attached to two SFP+ ports directly. The other two PE are attached to the same SFP+ port. Arbitration is done in a round-robin fashion.

The entry-point `pre-arch` is called before architecture generation and is used to determine if the desired configuration is valid for the given board. If more ports than available are requested, this stage aborts the bitstream generation process immediately.

The entry-point `post-address-map` is called after the address map of the rest of the system has been created. At this point, the plugin ensures that necessary control registers are accessible by the host.

The implementation itself is done by another feature of the plugin system. TaPaSCo can be extended by defining subsystems. A subsystem in TaPaSCo is automatically connected with the clock and reset system and can be used to provide additional functionality. The SFP+ plugin creates a subsystem called `Network` by creating a function called

```
1 proc create_custom_subsystem_network {{args {}}}
```

This function is called by TaPaSCo inside a fully generated subsystem environment and can be used to build the desired functionality. In this case, the plugin instantiates the necessary IP for the network interfaces and connects the PEs to them as desired in the configuration.

Part II

APPLICATION ACCELERATION USING FPGA

INTRODUCTION TO PART II

> The road to wisdom? – Well, it's
> plain and simple to express:
> Err
> and err
> and err again
> but less
> and less
> and less.
>
> Piet Hein

Part I introduced FPGAs as a method to accelerate applications. However, Chapter 1 and Chapter 4 described a variety of challenges that hinder the widespread application of FPGA. Hence, a reader might ask the question why he or she should deal with the hassle, and not use CPUs and GPUs instead. And they might be right, depending on the specific application. FPGAs are not a one-size-fits-all solution to every acceleration problem. If the FPGA is used to emulate a GPU, it will be slower than highly optimized ASIC GPUs. The same is true when using soft core processors on an FPGA, instead of using dedicated CPUs.

However, there are many applications that profit from the flexibility that FPGAs offer. If the FPGA is used to implement specialized accelerators that are optimized to their particular use case, they can provide significant speedups with a lower energy envelop, compared to general purpose hardware.

To demonstrate that FPGA can be highly competitive, this part of the thesis introduces FPGA based acceleration in four different domains.

The accelerator presented in Chapter 6 is a stereo-vision accelerator that is much faster and more efficient than GPU or CPU implementations of the same algorithm. Stereo vision deals with the extraction of depth information from stereoscopic images. The employed algorithm is Semi-Global Block Matching (SGBM) taht computes highly accurate depth maps. The computations require complex communication between the individual stages which limits parallelism on traditional off-the-shelf hardware. On the FPGA, however, the communication pattern can be efficiently implemented using systolic arrays.

The second domain that is discussed here is in-network processing (INP). Chapter 7 starts out by introducing the domain of in-network processing and shows how Bluespec can be used to efficiently deal with network traffic on FPGA.

The first INP accelerator, presented in Chapter 8, computes common distributed database operations directly in the network. The FPGA

acts as a network switch and performs stateful operations such as hash joins directly on the switch. This results in improved system performance and lowered bandwidth requirements.

The second INP accelerator, as described in Chapter 9, accelerates a consensus protocol in the network. Again the FPGA acts as the switch. The resulting accelerator brings down the latency of memory requests over the network far enough to make network attached main memory feasible.

Chapter 10 introduces a FPGA based Spiking Neural Network (SNN) simulation. The accurate and real-time simulation of neurons of the inferior-olive nucleus region of the brain enables novel ways for brain research without the need for in-vito experiments. In this case, the FPGA is a prime accelerator choice, as the communication scheme of the neurons can be implemented efficiently.

6

SEMI-GLOBAL BLOCK MATCHING

The work presented in this chapter has been published in:

1. Jaco Hofmann, Jens Korinth, and Andreas Koch. "A Scalable Latency-Insensitive Architecture for FPGA-Accelerated Semi-Global Matching in Stereo Vision Applications." In: *IEEE Proc. International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2016

2. Jaco Hofmann, Jens Korinth, and Andreas Koch. "A Scalable High-Performance Hardware Architecture for Real-Time Stereo Vision by Semi-Global Matching." In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. Best Paper Runner-Up. 2016

Allowing computers to perceive their environment is still one of the most challenging tasks in computer vision. Especially stereo vision, the perception of depth using two cameras, is important for many areas such as robotics and autonomous driving. Stereo vision uses two cameras that are located some distance apart horizontally, but are on the same level vertically. Pixels in the images captured by the two cameras are thus displaced only in the horizontal direction, with the maximum pixel offset (traditionally called *disparity*) limited by the distance of the two cameras. The computed disparity for the pixel can then be used to derive depth information from the stereo images using triangulation (pixels closer to the cameras have larger disparities).

The algorithm at the center of this work is called Semi-Global Block Matching (SGBM), which is one of the fastest algorithms also scoring well on accuracy in stereo vision benchmarks such as KITTI [39] and Middlebury [104]. This has led to its widespread use in many practical applications.

This work proposes a novel hardware architecture to compute the SGBM algorithm on FPGA. The parametrized architecture is highly scalable, easily allowing implementations on small low-power devices (e.g., in autonomous robots) as well as for large high-performance chips (e.g., in stationary use-cases for processing multiple high-resolution video streams). As shown in Figure 6.1, the focus lies in disparity computation; rectification and the actual camera interfaces will not be discussed.

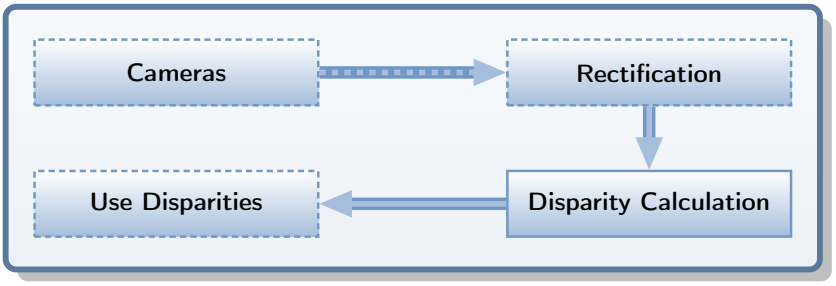


Figure 6.1: Typical stereo vision system. This paper focuses on the disparity calculation step.

6.1 SEMI-GLOBAL BLOCK MATCHING

The SGBM algorithm was introduced by Hirschmüller in 2005 [48]. It provides good accuracy at manageable computational effort and is robust with regard to choices for configuration parameters [49].

A key difference from the simple block-based approaches, which generally compute matches by aiming for minimal sums of differences between pixel intensities in the base and matching images, is the use of a more involved cost function. That cost function not only has a widened scope (considering not just individual pixels or local neighborhoods, but pixels along *paths* across the *entire* image), but also looks at higher-level characteristics (e.g., Mutual Information and Census), instead of pixel intensities.

The original implementation of the SGBM algorithm in [48] uses Mutual Information (MI) in the cost function. The advantage of using this characteristic, which was introduced by Viola and Wells in [121], is its quality even when faced with unrectified image data. However, it does not scale well to images having a greater depth (larger Z coordinates, especially beyond depths represented as 8 bit values).

Further research by Hirschmüller et al. identified Census as a robust matching cost calculation. It relies on encoding intensity *relations* (such as "darker than center pixel") between the pixels in a neighborhood in bit-vectors, and then determining the cost of a potential match based on the Hamming distance between a pair of these bit-vectors.

Here a variation of that idea is used, which considers the difference between straight *counts* of pixels fulfilling the relation instead of the Hamming distance as cost. This approach, called a *non-parametric rank transform* has a matching quality similar to Census, but is easier to implement for high compute performance [130].

The function $C(\mathbf{p}, d) \in \mathbb{N}_0$ is used to denote the cost of matching a pixel $\mathbf{p} = (x, y)$ at coordinates (x, y) in the base image at an assumed disparity (offset) of d at coordinates $(x - d, y)$ in the matching image. As suggested by [130], the differences in the counts of pixels darker than the center pixel (a parametric rank transform) can be used to realize C (see Eq. 4). Section 6.3 discusses these equations in more

detail. To determine the actual best match, these costs are calculated for all potential disparities $d < D_{max}$, where D_{max} is the upper limit of the potential disparity (due to the physical mounting distance of the two cameras). At first approximation, the match with the lowest cost is assumed to indicate the true disparity $\arg\min_{d<D_{max}} C(\mathbf{p}, d)$ between base and match images for an individual pixel $\mathbf{p}$ (but see below for further constraints).

To achieve better matching accuracy, (semi) global approaches such as SGBM compute these costs of potential matches not just between individual (or neighborhoods) of pixels, but along multi-pixel *paths* stretching across the entire image. The cost of matching along an entire path, described by the relative offset of path elements $\mathbf{r} = (\Delta x, \Delta y)$, for an assumed disparity d is denoted as $L_{\mathbf{r}}(\mathbf{p}, d)$. These paths are distributed evenly over the image (see Figure 6.2 for examples) for a global view of the matches. Typically, at least eight evenly distributed paths are used (Figure 6.2.b), but 16 are suggested for optimal coverage. The number and arrangement of paths has a direct impact not only on the matching accuracy, but also on the computational effort and, in this case, on the actual architecture of the SGBM hardware accelerator.

The goal is a compromise between performance and accuracy. As shown in [12], a reduction from eight down to the four paths 0° ($\mathbf{r} = (1, 0)$), 45° ($\mathbf{r} = (1, 1)$), 90° ($\mathbf{r} = (0, 1)$) and 135° ($\mathbf{r} = (-1, 1)$) (Figure 6.2.a) results in an accuracy loss of only 1.7% (increase in count of mislabeled disparities) in the Middlebury benchmark, but allows a highly efficient hardware architecture computing the $L_{\mathbf{r}}$ for all of these paths in parallel.

If even the limited accuracy loss is not acceptable, the proposed hardware architecture could be used to perform a second pass over the image, computing the remaining paths 180° ($\mathbf{r} = (-1, 0)$), 225° ($\mathbf{r} = (-1, -1)$), 270° ($\mathbf{r} = (0, -1)$) and 315° ($\mathbf{r} = (1, -1)$), starting from the opposite corner. This would reduce the frame rates for matching by half (see Section 6.4). As the paths in the selected arrangement are no longer distributed evenly across the image, some minor aliasing (non-isotropic) effects can be measured in some benchmarks, but these should not affect usability in real-world scenarios.

The raw cost $L'_{\mathbf{r}}(\mathbf{p}, d)$ for matching the pixels $\mathbf{p}$ along a path $\mathbf{r}$ for an assumed disparity of d is calculated using the formula

$$L'_{\mathbf{r}}(\mathbf{p}, d) = C(\mathbf{p}, d) + \min \begin{cases} L'_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, d) & .a \\ L'_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, d - 1) + P_1 & .b \\ L'_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, d + 1) + P_1 & .c \\ \min_i L'_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, i) + P_2 & .d \end{cases} \tag{6.1}$$

These raw path costs are calculated for all of the selected paths, for all potential disparities d up to the limit D_{max}. For each pixel, both

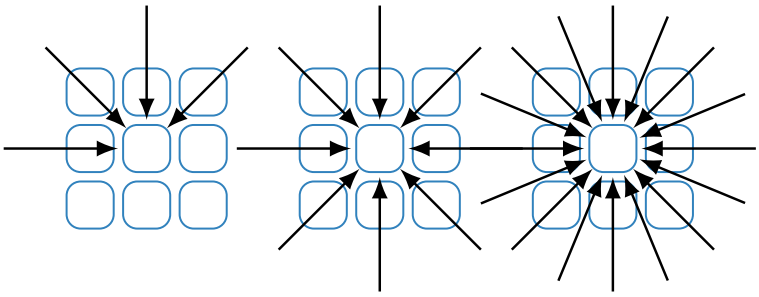


Figure 6.2: (a) Four, (b) Eight, and (c) 16 directions used in Semi-Global Block Matching.

the local cost C as well as a semi-global component is evaluated. The latter considers four characteristics observed in real-world images, of which the minimum is added to the local cost: The first characteristic is the cost of the *prior* pixel along the path (1.**a**), the second and third components penalize small disparity changes of $|\Delta d| = 1$ by P_1 (1.**b** and **.c**), while the final term (1.**d**) penalizes larger disparity changes (so-called *discontinuities*) by P_2. P_1 is usually determined off-line experimentally by analyzing input images typical for the actual stereo vision use-case. P_2, on the other hand, is adjusted dynamically at run-time: As disparity discontinuities are often also represented as pixel *intensity* changes, computing $P_2 = \frac{P_2'}{|I_{\mathbf{p}} - I_{\mathbf{p}-\mathbf{r}}|}$ compensates for different pixel intensities $I_{\mathbf{p}}$ and $I_{\mathbf{p}-\mathbf{r}}$ along the path $\mathbf{r}$. As for P_1, P_2' is a constant determined experimentally based on representative sample images offline. For further discussion of the path cost calculation, please refer to the original work by Hirschmüller [48]. To determine the (semi) global matching cost, the path costs are summed up across all paths. However, for a hardware implementation, it is worthwhile to consider a slightly changed formulation.

In hardware, a key characteristic from both the performance as well as area usage perspectives is the *word width* (in bits) of arithmetic operators and data types. Since the paths will run across the *entire* image, they can be quite long (depending on the camera resolution), and summing their costs can result in large values that need wide words for computation and storage. This can be counteracted by subtracting from the raw path costs for a pixel $L_{\mathbf{r}}'(\mathbf{p}, d)$ the *minimum* of the path costs for all assumed disparities d for the *prior* pixel $\mathbf{p} - \mathbf{r}$ along the path $\mathbf{r}$. The effect of encoding only the differences between prior and current pixels leads to a reduction of the magnitude of the values, which require correspondingly narrower data words for storage and computation. Thus, the hardware-optimized path cost computation becomes

$$L_{\mathbf{r}}(\mathbf{p}, d) = C(\mathbf{p}, d) + \min \begin{cases} L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, d) \\ L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, d - 1) + P_1 \\ L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, d + 1) + P_1 \\ \min_i L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, i) + P_2 \end{cases} - \min_j L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, j).$$

The path costs $L_{\mathbf{r}}$ along all paths $\mathbf{r}$ are summed up as $S(\mathbf{p}, d) = \sum_{\mathbf{r}} L_{\mathbf{r}}(\mathbf{p}, d)$. The disparity d with the minimal matching cost $\arg\min_d S(\mathbf{p}, d)$ is considered the winning disparity for the pixel $\mathbf{p}$. These winning disparities are output by the accelerator for each pixel, as input for later computing the actual depth (Z-axis position, not discussed here).

As indicated above, in practice additional constraints need to be imposed to clean-up outliers and mark invalid disparities: The result of $\arg\min_d S(\mathbf{p}, d)$ might be multi-element set, meaning that the minimal matching cost for a pixel $\mathbf{p}$ occurs for *different* potential disparities d. With such a non-unique cost, the algorithm cannot decide on a single winning disparity, and instead registers the disparity for this pixel as "invalid".

Additionally, a so-called *left/right check* is performed, which compares the results of the algorithm when running it with swapped roles of base and match images. This check can also be implemented efficiently (avoiding recalculating all disparities for the former match image now used as base) by re-using the previously computed $S(\mathbf{p}, d)$ along an epipolar line as $\arg\min_d S((x(\mathbf{p}) + d, y(\mathbf{p})), d)$ to select the winning disparity d for the second image. The left/right check sets the disparity to "invalid" if the corresponding disparities of the original and role-swapped passes differ by more than one. This step eliminates phantom disparities resulting from occluded surfaces that are visible in one image, but hidden in the other.

Finally, the disparity map is post-processed using a basic 3×3 median filter to suppress outliers.

6.2 RELATED WORK ON HIGH-PERFORMANCE SGBM IMPLEMENTATIONS

Several implementations of SGBM exist for a wide variety of use-cases. However, they often have unsatisfactory performance or high power requirements.

Even when exploiting current vector extensions (SIMD) such as Intel AVX2 on a fast i7-4960HQ processor, a recent software implementation [116] achieved only 16 frames-per-second (fps) for VGA image pairs and $D_{max} = 128$. This processor is rated to draw 47 W under load.

The less power-hungry software solution described in [6] targets the P4080 embedded eight-core processor. Clocked at 1.2 GHz, the implementation achieves 0.5 fps on VGA images, but limits the search

space to $D_{max} = 64$. The P4080 is rated to draw less than 30 W even when fully loaded.

A different trade-off was used for a heterogeneous embedded system in [107]: By combining a small Core2Duo system with an embedded OMAP3530 ARM processor and a Xilinx Spartan 6 FPGA, it achieved 14.6 fps, but has a latency of 250 ms for processing a single image of 1024×508 pixels and $D_{max} = 128$. This latency might be too high for certain real-time use-cases.

The use of GPUs leads to results similar to that of CPUs. [84] describes an implementation achieving 11.7 fps on an NVidia GTX480 GPU for VGA images with $D_{max} = 64$. However, this GPU is rated to draw between 250 *W* and 300 *W* of power when loaded. Another GPU based implementation can be found in [47]. The paper presents an implementation of SGBM on an embedded Tegra X1 GPU, achieving 19 fps for VGA images and 128 disparities. They also run their code on a Titan X which is able to process 237 fps.

FPGA implementations do significantly better, both with regard to performance as well as power efficiency: In [122] an architecture is proposed that is capable of processing 1024×768 pixels with $D_{max} = 96$ at 31.79 fps on a Altera EP4SGX230 device. Another approach is followed in [99], where high-level synthesis from C to hardware is used to explore different strategies targeting the Xilinx Zynq Z7020 system-on-chip on a ZedBoard. They achieve 30 fps at VGA resolution, but have a tight search limit of $D_{max} = 16$. A combination of FPGA and CPU processing is used in [56] to achieve 60 fps for images of 752×480 pixels on a Xilinx Artix 7 FPGA. While exact power numbers are not given, in many cases FPGA draw less than 10 W for computing purposes when the high-speed serial transceivers are not used.

6.3 ARCHITECTURE

The architecture proposed here is inspired by prior work by Banz et al., presented in [12]. However, not only has the architecture been enhanced by introducing an extra level of fine-grained parallelism (e.g., parallel disparity computation and sorting), it has also been implemented using a state-of-the-art latency insensitive design style in a next-generation hardware description language. As a result, it is significantly more scalable, easier to extend, yet also much faster than the original work (even when compensating for the differences in FPGA target technologies).

The algorithm described in Section 6.1 is mapped to the structure shown in Figure 6.3. The modules in Stage 1 have the base and/or matching image as inputs. The cost module implements the computation of the per-pixel (neighborhood) cost (e.g., using Census or Mutual Information) from both input images. This module outputs the cost $C(\mathbf{p}, d)$ for all pixels and all disparities up to D_{max} as a

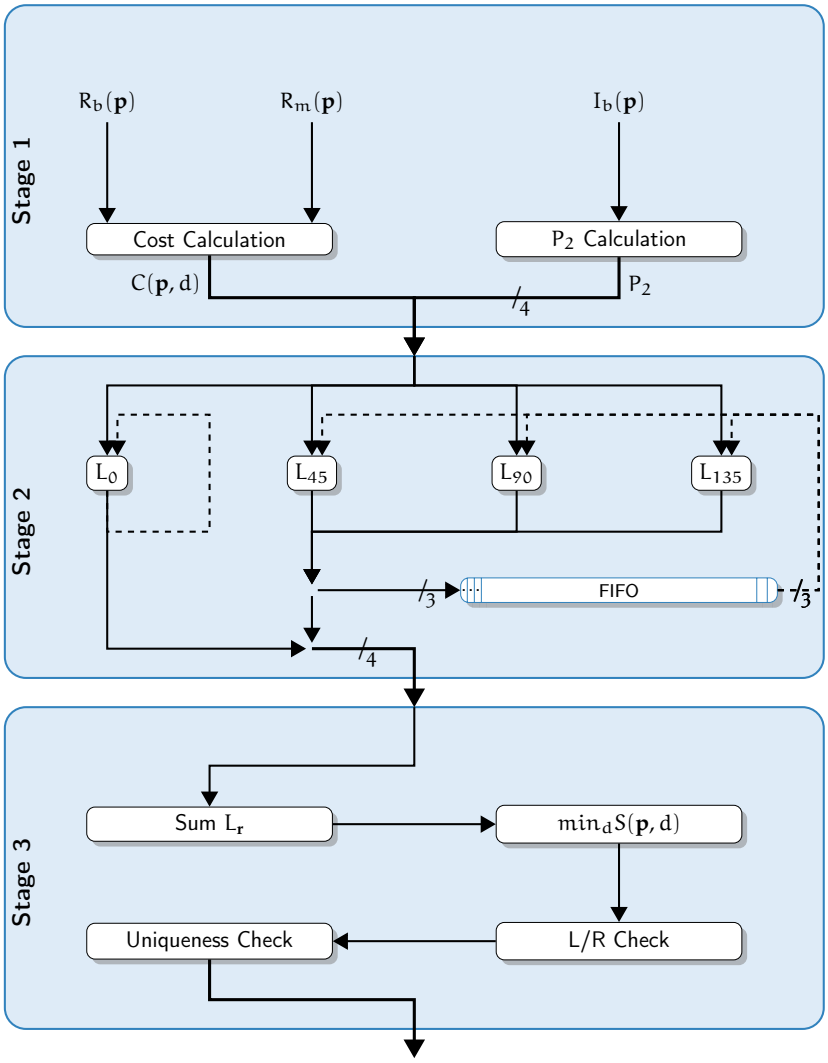


Figure 6.3: Base architecture for SGBM. Stage 1 produces the per-pixel costs and P_2 penalty values. Stage 2 calculates the path costs, using the outputs of the previous stage as well as prior paths costs stored in a stage-internal buffer. Stage 3 computes the disparities from the path costs.

stream starting from $d = 0$ up to $d = D_{max} - 1$. Secondly, Stage 1 computes the P_2 values (intensity-based penalty for discontinuities), which are generated from the base image for every pixel and the four path directions used. Thus, it outputs a stream of four P_2 values per pixel. Only Stage 1 actually reads image data, the next stage operates only on streams of per-pixel costs and P_2 values.

From these streams, Stage 2 computes the disparities along the four paths in parallel, using a separate module for each path $\mathbf{r}$. Since this computation requires not just the *current* per-pixel cost $C(\mathbf{p}, d)$, but also prior costs $L_{\mathbf{r}}(p, d)$ from *earlier* locations along the paths, the stage needs internal feedback, using buffers to delay the earlier values appropriately (storing them as three-element vectors). In summary, the Stage 2 modules require $C(\mathbf{p}, d)$, which is constant for all paths, P_2 which is constant for each pixel, $\min_i L_{\mathbf{r}}(\mathbf{p} - \mathbf{r}, i)$ (for the P_2-penalized L term and the magnitude reduction), which is constant for all disparities of a pixel, and finally $L_{\mathbf{r}}(\mathbf{p}, d-1)$, $L_{\mathbf{r}}(\mathbf{p}, d)$ and $L_{\mathbf{r}}(\mathbf{p}, d+1)$ (for the L terms penalized by P_1), all of which are unique for all disparities and all pixels and all paths. The stage produces streams of four-element vectors containing the costs for the four paths for each pixel, with the path costs for the different potential disparities $0 \leqslant d < D_{max} - 1$ just being streamed-out in order.

Stage 3 operates on these streams to compute $S(\mathbf{p}, d)$ for each potential disparity and determine $\min_d S(\mathbf{p}, d)$. The latter is then checked for uniqueness and also undergoes the left/right check re-using the previously computed costs (see Section 6.1 for details). The output of the module is a stream of the final winning disparities, cleaned-up by a 3×3 median filter, or the "invalid" markers for pixels for which no winning disparity could be determined.

This base architecture can already perform the complete SGBM computation, and, with some care in the implementation, can be fully pipelined. However, it performs only the computation of the path costs $L_{\mathbf{r}}$ in parallel. This can be improved both at the fined-grained as well as at the coarse-grained levels.

FINE-GRAINED PARALLELIZATION One key contribution of this work beyond [12] is the computation of per-pixel and per-path costs for n potential disparity values in parallel (see Figure 6.4). Thus, Stage 1 now needs to compute n per-pixel costs, which are then streamed to Stage 2 as n-element vectors. Note that the P_2 computation is unaffected, it is still performed only once per path for each pixel. Considerable extension is required in Stage 2, which will now be replicated n-times to accept the n-element vectors from Stage 1 and process them in parallel. Two areas require special care: The intra-stage buffers are becoming larger (now having to hold $n + 2$-element vectors) with a more complex forwarding network, also the calculation of the minimal $L_{\mathbf{r}}$ values has to happen in parallel (achieved by a fast

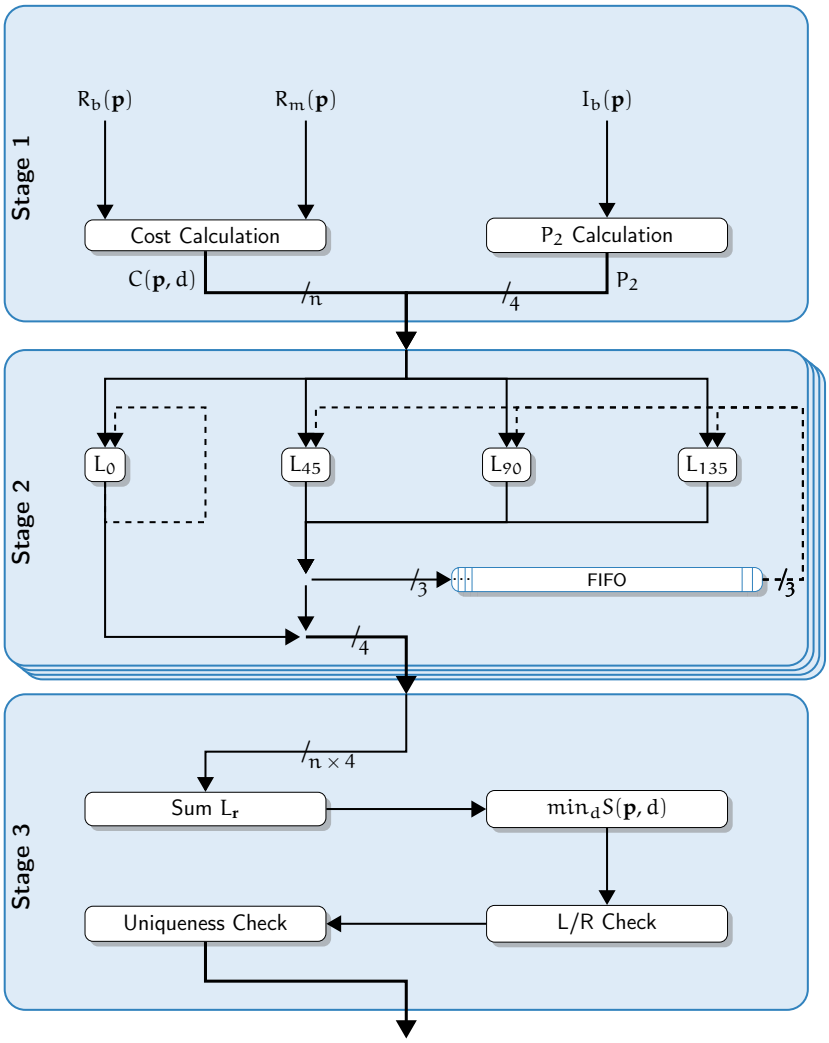


Figure 6.4: Extending the base architecture with fine-grained parallelism: Parallel cost computations (per-pixel, per-path) for multiple potential disparity values

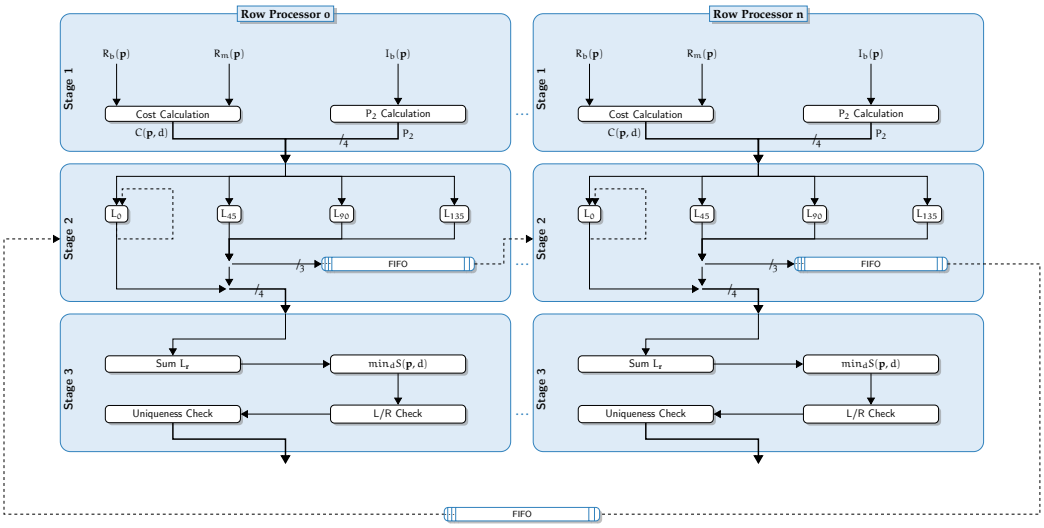


Figure 6.5: Coarse-grained parallelization: Processing multiple rows in parallel

comparator tree). Stage 3 is similarly sped-up, replacing the sequential summing of paths for each potential disparity value with parallel adder trees. The critical path determining the performance of this solution is the parallel computation of $\min_i L_{0°}(\mathbf{p} - \mathbf{r}, i)$ across all potential disparity values i, as its results are immediately needed to compute the costs for extending the path to the next pixel. Note that the other $L_{\mathbf{r}}$ units' outputs are buffered and required only in the next row, but $L_{0°}$ has a self-loop.

COARSE-GRAINED PARALLELIZATION The base architecture can also be parallelized on a coarse-grained level (Figure 6.5). This requires m multiple instances of the entire base architecture (called a *Row Processor* in this context). Each Row Processor is responsible for processing one line of the input images, leading to an image stripe m rows in height being processed in parallel. Buffers between the Row Processors forward the intermediate data from the $L_{\mathbf{r}}$ computations in Stage 2 to Stage 2 in the next Row Processor, where they will be consumed by the corresponding $L_{\mathbf{r}}$ units. Note that this forwarding occurs only for paths at 45°, 135°, 90° angles, as the path at 0° does not require data from the prior row (see Figure 6.2.a). These forwarding buffers also provide the wrap-around to the first Row Processor, as there will be fewer Row Processors than image rows, and the first Row Processor (which will be processing the first line of the next stripe down) depends on the last Row Processor (which was processing the final line of the previous stripe up). The performance of this approach depends on how quickly the inter-row buffers can be filled with data, as Row Processors waiting for data from their predecessors will remain idle. Similarly, a Row Processor will stall if it cannot deposit its output due to the corresponding buffer to next row being full. This

coarse-grained approach was already suggested in [12]. By treating it as a wavefront array (systolic array with handshake instead of lock-step data propagation), it lends itself ideally to an implementation in the new latency-insensitive hardware design style applied.

Both parallelization techniques can be combined to mitigate their respective weaknesses (critical path length in the fine-grained approach, Row Processors stalling in the coarse-grained one). The complete architecture with both forms of parallisation is shown in Figure 6.6. Automatic design space exploration is used in Section 6.4 to derive the optimal composition of the two approaches.

SELECTED ARCHITECTURAL DETAILS As discussed in Section 6.1, thethe non-parametric rank transform variant of Census is used. The rank transform consists of counting the *number* of pixels in a neighborhood that are of lower intensity than the center pixel:

$$R(\mathbf{p}) = \left\| \mathbf{p}' \in N_s(\mathbf{p}) | I(\mathbf{p}') < I(\mathbf{p}) \right\|, \tag{6.2}$$

Here, $R(\mathbf{p})$ is the rank transform of pixel $\mathbf{p}$, $N_s(\mathbf{p})$ the neighborhood around pixel $\mathbf{p}$ encompassing all pixels with a distance less or equal to $(s-1)/2$ from the center, and $I(\mathbf{p})$ the intensity of pixel $\mathbf{p}$. The neighborhood, also called a kernel, is square with a total size of s^2 pixels. Each rank transformed value is encoded in $\lceil \log_2 (s^2) \rceil$ bit.

The calculation of $R(\mathbf{p})$, which serves as input to the cost calculation stage, is done in parallel for both images. The Row Processors are fed with streams of $R_b(\mathbf{p})$ for the base image and $R_m(\mathbf{p})$ of the matching image, as well as a stream for the P_2 penalty values. If more than one Row Processor is present in the system, round-robin is used to distribute the inputs one row at a time (leading to processing occurring in the downward moving stripe). All Row Processors require input FIFOs which can store a full row to enable operation as a wavefront array.

The actual per-pixel matching cost computation based on the rank transform R is thus

$$C(\mathbf{p}, d) = \| R(\mathbf{p}) - R((x(\mathbf{p}) - d, y(\mathbf{p}))) \|. \tag{6.3}$$

The wrap-around buffer from the last to the first Row Processor (shown in Figure 6.5) is larger than the normal inter-Row Processor buffers, as it has to hold all D_{max} intermediate results for every pixel in the last row.

In a design with multiple Row Processors, the calculated disparities are buffered in FIFOs until they are retrieved by the output module. The output module then merges the outputs of the Row Processors using the same round-robin scheme as the input module. The 3×3

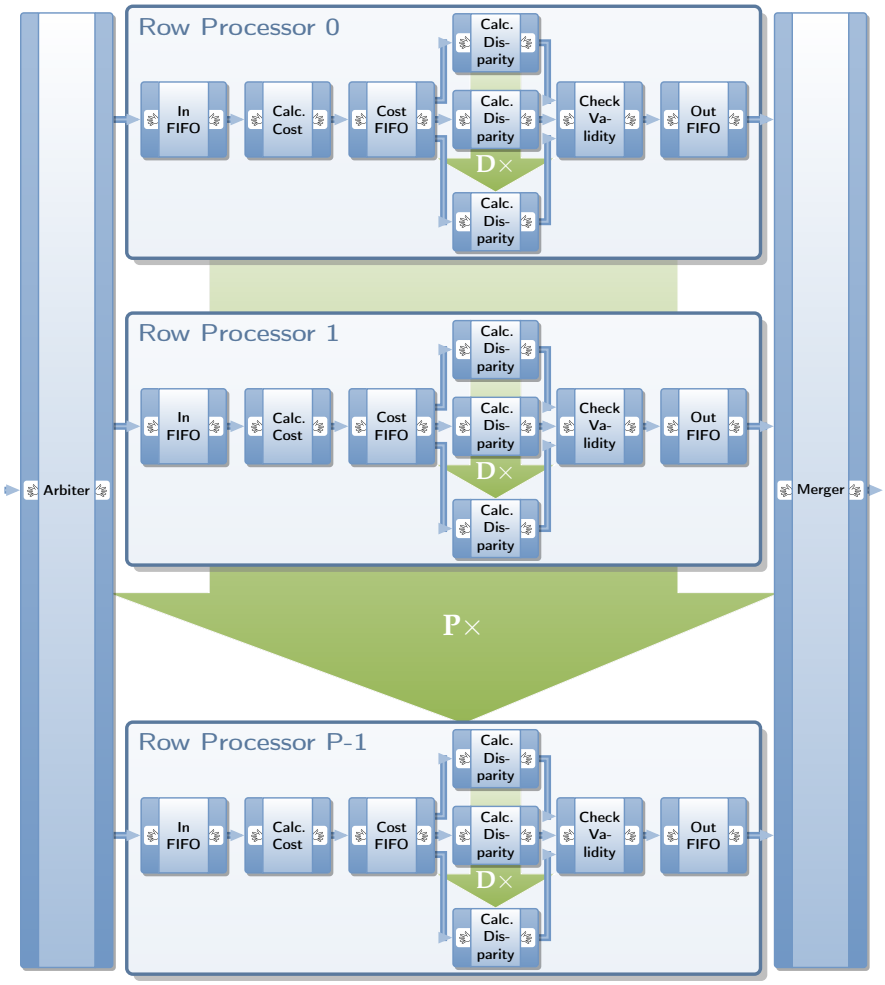


Figure 6.6: Detailed view of the SGBM calculation sub-architecture. Each row processor receives a line of the rank-transformed input image in sequential order. The row processor calculates all disparities for its specific row. The semi-global data from other input lines, as shown in Equation (6.2), is distributed in a systolic array-pattern. Two methods of parallelization are employed: 1) Using multiple row processors, sequential lines of the input image can be processed in parallel. 2) Finer grained parallelism is employed inside of each row processor, where multiple disparities are calculated in parallel.

median filter is then applied to the merged stream remove outliers in the calculated disparities.

As shown by Hirschmüller in [48] the maximum value of any L is always less than $C_{max} + P_2$. This limits the word widths required for data storage and arithmetic operators in the hardware implementation.

INFRASTRUCTURE The integration of the accelerator into TPC requires certain interfaces to allow communication with the outside world. As shown in Figure 6.7, the architecture has two AXI Master interfaces to talk to the main memory. The interfaces are used to read both input images in parallel. One of the interfaces is additionally used to write back the disparty map. Another interface is used to interface the host, controlling the execution, with the accelerator. Parameters such as the addresses of the input and output images as well as their size can be set before execution. Lastly, an interrupt signal signals execution completion to the host.

6.4 EVALUATION

The approach is evaluated at three levels: The first consideration is accuracy, then the simulated target-independent performance of the hardware architecture in terms of clock cycles, and finally the wall-clock performance on three actual FPGA platforms, encompassing embedded system and data center use.

6.4.1 *Accuracy*

Since the core algorithm is the same as that of [12], the same disparity computation accuracy is achieved. Using the Middlebury[104, 106] benchmark, the algorithm produces on average 8.4 % disparities exceeding an error threshold of one pixel. The difference between the results of the proposed architecture and the ground truth in non-occluded areas is 9.5 % for Cones, 13.3 % for Teddy, 6.8 % for Tsukuba, and 4.1 % for Venus. A sample result for the Teddy test set is shown in Figure 6.8. If required, higher accuracy could be achieved by doing a full left/right check (full computation instead of re-using epipolar projected values), or performing two passes over the image to use eight paths. Both of these approaches approximately halve the performance of the accelerator, but it would remain useful even then for real-time processing at 30 fps or above (see below).

6.4.2 *Platform-independent performance*

Cycle-accurate simulation of the architecture (described in highly parametric Bluespec SystemVerilog, BSV) was used to determine the run-time characteristics of sample implementations. Comparison to

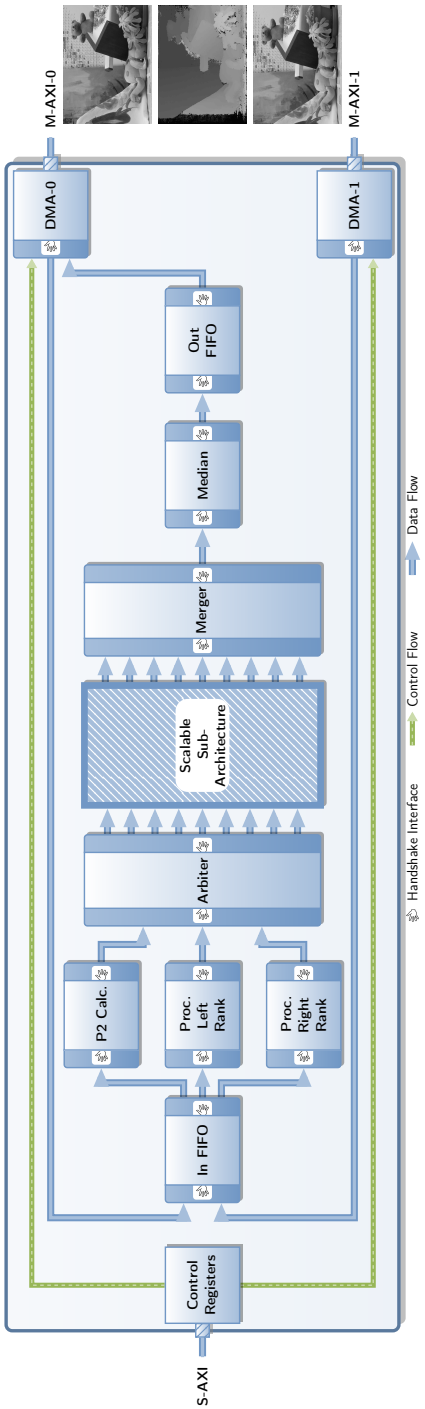
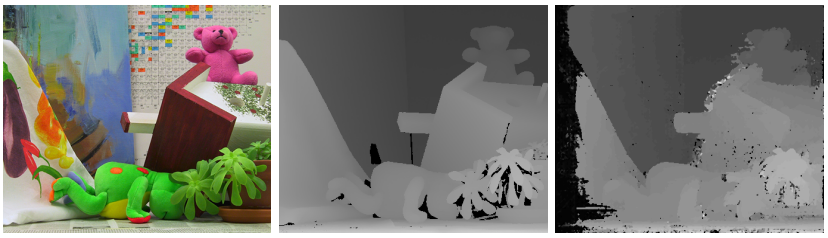


Figure 6.7: Operation of the proposed architecture. The images are read from host memory by two DMA engines. A control interface is used to alter the behavior of the algorithm. The base images are processed by rank transform and P_2 calculation cores. The results are then fed into a parallel sub-architecture which performs the actual SGBM calculations. The resulting disparities are filtered to suppress outliers, and forwarded to one of the DMA engines for transfer back to the host.



(a) Teddy image set left image

(b) Teddy image set ground truth

(c) Disparity produced by the proposed architecture

Figure 6.8: Disparity comparison for the Teddy image set.

the actual hardware implementations (Section 6.4.3) shows, that these simulations are actually representative of final performance.

The core is evaluated at three image resolutions: 640×480 pixels (VGA), 1280×720 pixels (720p) and 1920×1080 pixels (1080p). VGA resolution images are evaluated at $D_{max} = 64$, the higher resolutions have $D_{max} = 128$. For all resolutions, the number of clock cycles needed to complete a single frame are determined through simulation.

The examination contains different compositions of coarse- and fine-grained parallelism. An implementation is described by the pair (**#p**, **#d**), which indicates the use of **#p** Row Processors, each computing **#d** assumed disparities in parallel. For each of the resolutions, automatic design space exploration is used to generate 250 implementation alternatives, shown on the X-axis in order of increasing area or performance. Due to space constraints, only a subset of the alternatives could be labeled here with (#p,#d).

As shown in Figure 6.9 for VGA images, the architecture scales well with increasing the number of Row Processors, down to a lower bound of 654644 cycles, at which point a single pixel is calculated in 2.11 cycles and a single disparity requires 0.033 cycles. This design is limited by the speed the input buffers can be filled, which could be increased even more by also applying fine-grained parallelization techniques to the Stage 1 computations (see Section 6.5).

At an assumed clock rate of 200 MHz (which is realistic, see Section 6.4.3) the architecture would reach up to 306 fps as shown in Figure 6.10. Such large and fast systems could be used to calculate the disparities over multiple cameras to produce a surround-view of a scene.

More interesting for low-power applications is the *minimal* frequency necessary to achieve 30 frames per second (a typical requirement for real-time processing). As shown in Figure 6.11, the architecture is able to fulfill this requirement at a frequency as low as 30 MHz for VGA images.

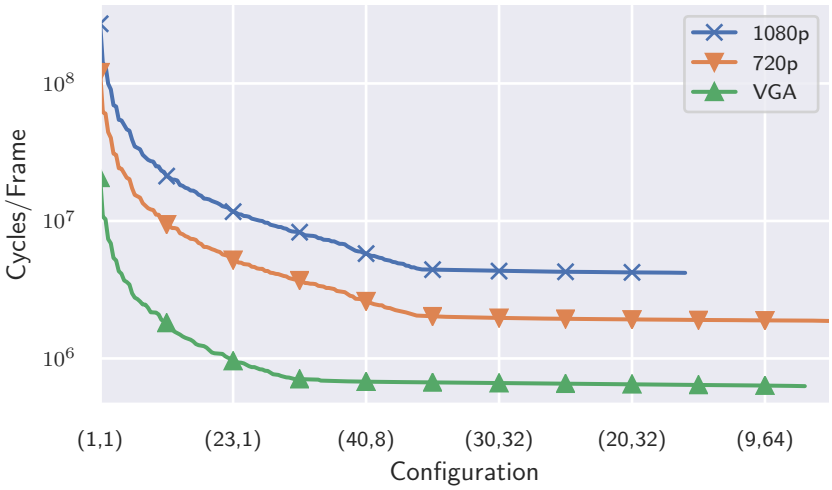


Figure 6.9: Cycles needed to process a single disparity map for varying degrees of parallelism.

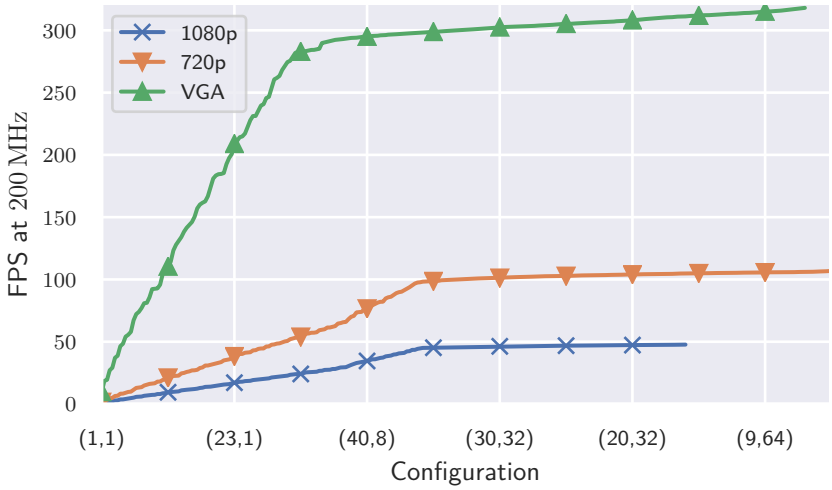


Figure 6.10: Frames per second achieved by the proposed architecture at a clock frequency of 200 MHz.

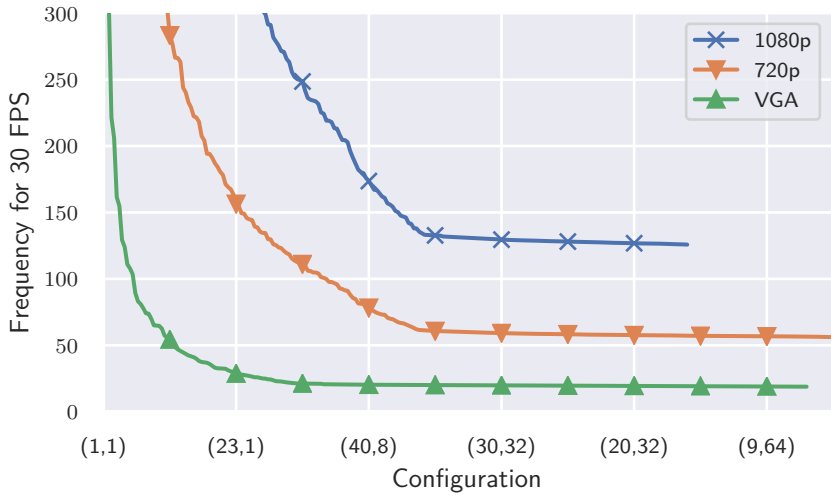


Figure 6.11: Frequency needed to achieve 30 frames per second on the proposed architecture for varying degrees of parallelism.

6.4.3 *Performance on real FPGA platforms*

The true performance of this approach can be measured only when actually mapping an implementation of the architecture to a real hardware platform, of which three cases are considered: ZedBoard (ZC7Z020T), Xilinx ZC706 (ZC7Z045) and VC709 (ZC7VX690T) development boards. The first two use Zynq-7000 reconfigurable system-on-chip devices, the last a large Virtex-7 FPGA.

On the tools side, Bluespec 2015.09 beta2 was use to compile the BSV descriptions into synthesizable RTL-style Verilog. TPC 2016.03 [68] was used to assemble the hardware accelerators into full system-on-chips (e.g., adding memory and control interfaces), and perform automatic design space exploration. The entire hardware system was then mapped to the FPGA devices, using Xilinx Vivado 2015.2 for logic synthesis, placement and routing.

Figure 6.12 shows the resource requirements of the core at VGA resolution for different degrees of parallelism. Only the smaller two platforms are considered here, as the large VC709 is severely underutilized at VGA resolution (it easily holds even the largest sensible VGA accelerator). The smaller Zedboard is capable of running a core with #p=8 row processors and #d=1 parallel disparities as the largest core, achieving 33 VGA fps. However, design space exploration by TPC has discovered that an SGBM core parametrized as (#p=5,#d=2) actually performs better (40 fps) at the 100 MHz clock frequency used on the Zedboard, and is also smaller. The ZC706 is capable of housing much larger cores: TPC exploration suggests core configurations of (21,1), achieving 140 VGA fps at 210 MHz, and (13,4) achieving 134 fps at 150 MHz.

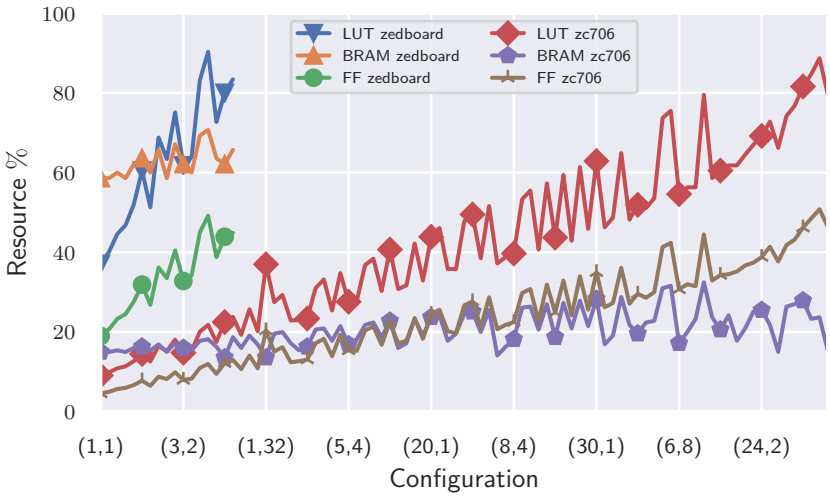


Figure 6.12: Hardware synthesis results for accelerators at VGA resolution for Zedboard and ZC706 platforms

The increase in image resolution and disparity range to 720p resolution and $D_{max} = 128$ grows the search space by six times. However, the implementations still manage to process images in real time. on the two larger platforms (the Zedboard is too limited for the higher resolutions). The ZC706, running at 140 MHz, is capable of supporting a (16,4) configuration which yields 32 fps. The larger VC709 handles up to 45 fps at 130 MHz using a (34,4) configuration.

The largest images tested are in 1080p resolution with $D_{max} = 128$, requiring yet another 2.25x increase in search space over 720p. Despite this large search space, the VC709 is again capable of real-time performance at 30 frames per second in a (20,4) configuration at 130 MHz. The smaller ZC706 tops out with a (12,4) configuration running at 140 MHz, yielding 12 fps.

In addition to performance and area, the energy consumption is an important characteristic when evaluating hardware accelerators targeting low-power use-cases. As can be seen in Figure 6.15, the architecture requires as low as 8.404 mJ to process a single frame. Slowing the clock frequency to reach a target of 30 fps results in similar energy requirements per frame as shown in Figure 6.16. The lower clock frequencies are counteracted by the longer active time per frame.

6.4.4 *Design Space Exploration in TPC*

The previous section evaluates the largest core fitting on a given platform. TPC, however, is capable of finding the configuration with the best performance for a given platform. This feature, called Design Space Exploration (DSE), takes into account the complex relationship

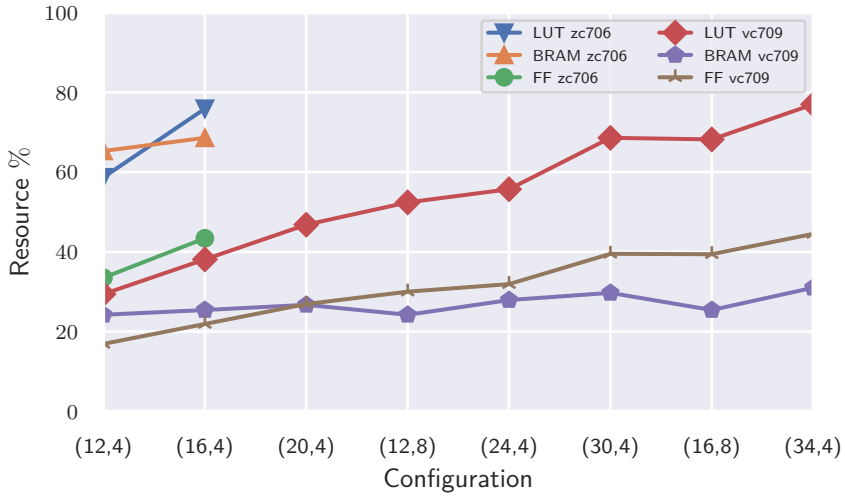


Figure 6.13: Hardware synthesis results for accelerators at 720p resolution for ZC706 and VC709 platforms

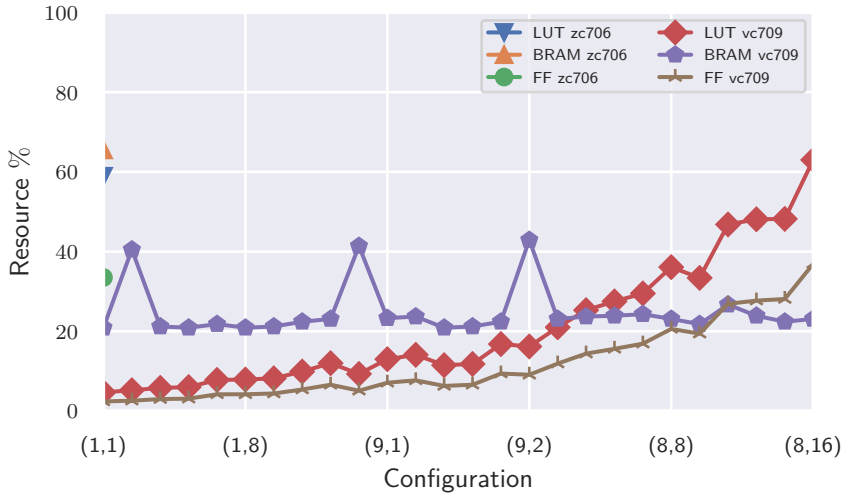


Figure 6.14: Hardware synthesis results for accelerators at 1080p resolution for ZC706 and VC709 platforms

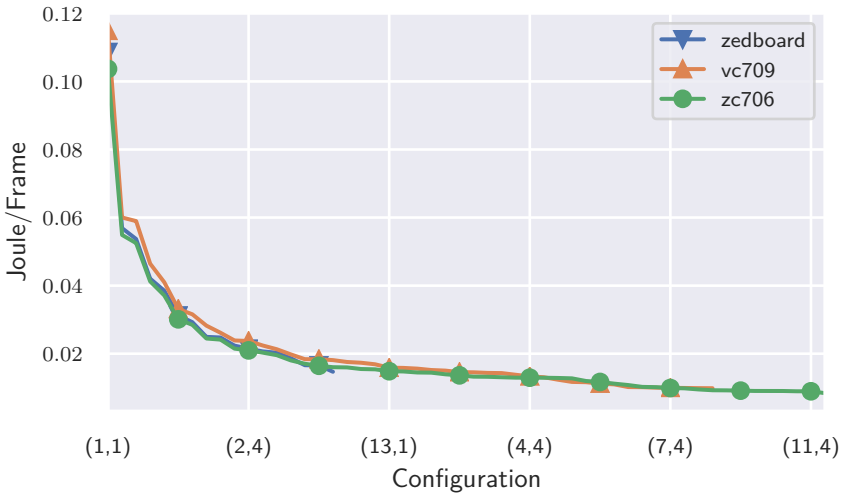


Figure 6.15: Energy consumed by different configurations of VGA-resolution SGBM accelerators at maximum fps

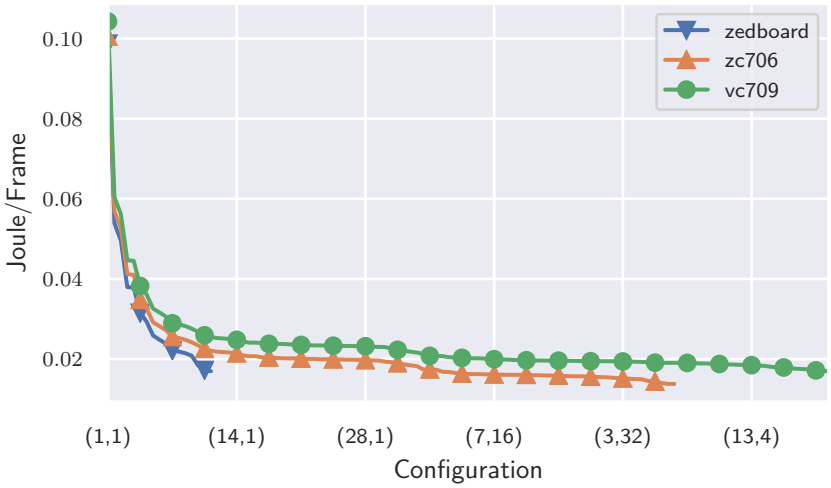


Figure 6.16: Energy consumed by different configurations of VGA-resolution accelerators when achieving 30 fps

between frequency, hardware utilization and parallelism. For example it might find that a core with less internal parallelism can run at a much higher frequency which results in an overall greater performance. On the other hand it might find that multiple smaller PEs perform better than one large PE.

The DSE step in TPC requires reasonable estimates of a given cores performance. The results in Section 6.4.3 indicate that the number of cycles per image calculated through simulation closely correspond to the real world performance of a given core. Accordingly, the simulation results were chosen as input to the DSE step.

Table 6.1 shows the results for each device and resolution, i.e., the designs with the highest heuristic score that achieved timing closure. The column **N** denotes the number of parallel instances of the core in the design, the column **F** the achieved design frequency in MHz. Using these designs, the performance on real hardware is evaluated: A C++ program using TPC-API, which can be compiled for all three platforms without changes to the source code, uses the accelerator pool to compute disparity maps for random images. The actual throughput achieved here (in frames per second) is shown in the last column **FPS** and corresponds closely to the throughput predicted as **h**-value by the heuristic. While accelerators operating at HD resolutions could not be placed on the ZedBoard, even that small platform can handle real-time ($>$ 25 fps) stereo vision computations at VGA resolution. The large VC709 board allows live processing of up to three independent 720p video stream pairs, or live processing of a single full HD 1080p stream pair.

Some of the optimal solutions found by DSE might easily have been overlooked in a manual approach: The best design for the VC709 and 1080p resolution was the result of the target clock frequency being *lowered* after a timing failure. Also, casual experimentation might lead a designer to go for higher values of P and lower values of D, as that leads to higher clock frequencies (e.g., 205 MHz for VGA resolution the ZC706 board). However, the best solutions for 720p and 1080p actually increase D and (for 1080p) lower P, and achieve the best throughputs despite running at lower clock rates.

6.5 CONCLUSION AND FUTURE WORK

The proposed architecture to compute semi-global matching on FPGA performs well over a wide range of scenarios. Low-power VGA configurations run at 30 fps with a clock as low as 74 MHz even on small FPGA such as that used on the Xilinx ZedBoard. For higher performance needs, the architecture offers multiple levels of parallelism, and can be tuned by TPC in an automatic design space exploration to discover optimal configurations.

Table 6.1: Design Space Exploration results as generated by TPC for the SGBM accelerator. Column P is the number of parallel row processors per accelerator, D specifies the number of parallel disparities per accelerator and N specifies the number of accelerators that are instantiated in parallel through TPC. F is the frequency achieved for the configuration after synthesis. Column h is the estimated performance based on a heuristic and FPS the performance achieved on the platform.

Platform	Resolution	P	D	N	F	h	FPS
zedboard	640×480	5	1	1	110	26.6	26.6
zc706	640×480	21	1	1	205	198.2	197.0
vc709	640×480	12	2	3	131	426.9	410.0
zedboard	1280×720						
zc706	1280×720	27	2	1	145	59.4	59.3
vc709	1280×720	20	2	2	121	75.8	74.9
zedboard	1920×1080						
zc706	1920×1080	17	4	1	140	23.3	23.3
vc709	1920×1080	13	8	1	122	28.6	28.4

The introduction of fine-grained parallelism into Stage 2 allows a much better adaptation of the accelerators to the needs of the individual use-case, as just increasing the number of Row Processors (as done in [12]) does not always result in the most efficient implementation.

Areas for future work include extending the use of fine-grained parallelism to Stage 1 of the architecture, namely the per-pixel cost computation (including the rank transform) and the P_2 calculation, as well as reducing the number of stalls in the Row Processor wavefront array by improvements in the buffering scheme.

7

SYSTEM-ON-CHIP INFRASTRUCTURE FOR IN-NETWORK PROCESSING

In-Network Processing (INP) in general gained a lot of traction in recent years with the rise of programmable switches. These switches, based for example on the Barefoot Tofino architecture, are capable of processing and switching 12.8 Tbit/s of network traffic. These programmable switching chips are utilized in a variety of other manufacturers switches and slowly increase adoption rate. Switches like these are mainly programmed in DSLs and through protocols such as P4[17] and OpenFlow[82]. In academia, many possible applications are discussed. Data Aggregation and Filtering is a natural task for on-switch processing and can be used in many applications including Parallel Databases, Machine Learning and Systems Monitoring. Other research focuses on protocol implementation inside the network. Protocols that benefit from very low latency, such as Consensus protocols[28], are prime examples. A discussion of different research questions inside the INP domain is presented in Section 7.2.

However, the current generation of programmable switches has some key limitations which makes them less suitable for many other applications: (1) Switches do not have a large amount of memory for state. (2) Switches can not generate new packets but only react on incoming packets. (3) Latency is high compared to the packet transmission itself. (4) Reconfiguration requires a restart of the switch. Solving these problems in next generation switching ASICs is an ongoing research topic in industry and academia.

In the meantime FPGAs, provide a good alternative for experimenting with in-network acceleration of applications. Compared to dedicated switching chips there are mainly two drawbacks: (1) The raw performance and port count of FPGAs is lower, (2) Programming requires specialized hardware knowledge. Problem 1 results from the lower clock frequencies of FPGAs and the limited number of available serial transceivers for ports. Processing multiple 100 Gbit/s, however, is very much possible [37, 87]. The second problem can be somewhat alleviated by utilizing HLS-tools such as SDNet. While these tools are easier to use, they come with a plethora of problems such as high latency and difficult integration into the FPGA boards.

This chapter presents two accelerators developed in the scope of INP and shows that FPGAs are a great tool to explore novel architectures and feasibility to define features needed by next generation switching ASICs. Before describing the accelerators themselves, a short intermezzo discusses how packet parsing on FPGA can be realized using Bluespec.

Afterwards, an accelerator for hash joins located inside a switch is presented. Finally, an implementation of a consensus protocol on FPGAs is compared to one running on Barefoot Tofino and a software implementation.

7.1 NETWORK PACKET PROCESSING IN BLUESPEC ON FPGA

The applications presented in the rest of this chapter rely on the FPGA to process incoming and outgoing network packets. Accordingly, the FPGA has to be programmed to parse incoming network packets. This process is usually problem dependent and the corresponding code parts have to be rewritten anytime the protocol has to be changed. This section presents a different approach that utilizes the high level functionality of Bluespec to separate the packet parsing from the functionality. The resulting packet parser generator improves productivity and simplifies changes to the underlying protocols.

The packets arrive at the accelerator as a stream of 64 bit words at a frequency of 156.25 MHz. Each packet is send as a stream and the last beat of the packet is denoted using the AXI4-Stream last signal. Although, AXI4-Stream uses a handshake mechanism (See Chapter 3 about handshaking), the stream receiver needs to accept any packet. Failing to do so can lead to dropped packets or loss of synchronization. Hence, the parser needs to ensure that no packets are dropped. Luckliy, the Bluespec compiler can be used to ensure that this is always the case.

Furthermore, the parser has to maintain a low latency profile. For instance, certain packets can be dropped right away because they are not targeted at the FPGA. Additionally, the packet should be processed as soon as possible in a cut-through fashion. The parser should not wait until the complete packet is available. This would neither be practical from a latency standpoint, nor be a good use of the memory on the FPGA.

The parser is implemented as a Bluespec module:

```
1 module mkPktParser
2         #(List#(PktParserStages#(pktType)) parserStages)
3         (PktParser#(pktType))
```

The module takes a List of `PkgParserStages` as argument. The logic of what happens after each part of the packet has been received is provided by the user of the module. The type `pktType` can be user defined and is usually the type that represents the Ethernet frame or parts of it, for instance the header. The `pktType` is automatically filled with more and more data as soon as it is available and is accessible by the parser stages. The interface that connects the module to the outside world contains three connections:

```
1 interface PktParser#(type pktType);
2     interface Put#(AXI4_Stream_Pkg#(64, 0)) pktStream;
```

```
3       method Action resetProcessing();
4       interface Get#(Bit#(64)) sideband;
5   endinterface
```

The `pktStream` interface connects the module to the Ethernet frame stream. The method `resetProcessing` can be used to reset the state machine that is implemented inside the module, for example, after synchronization has been lost. Lastly, the `sideband` interface can be used to extract raw stream data. This is useful if the `pktType` contains only part of the complete packet, for instance the header, and the payload is processed subsequently.

The parser is built from parser stages. Each stage is represented as a structure containing the necessary information for the parser to determine when this particular functionality should be called:

```
1   typedef struct {
2       Integer stage;
3       PktParserFun#(pktType) fun;
4   } PktParserStages#(type pktType) deriving(Eq);
```

First of all, the stage has to specify for which part of the packet it is relevant. For example, putting zero as the `stage` calls the function `fun` when the first beat of a new packet arrives. The function itself can be of two different types:

```
1   typedef union tagged {
2       ParseFunction#(pktType) FunctionSimple;
3       List#(ParseFunction#(pktType)) FunctionList;
4       } PktParserFun#(type pktType) deriving(Eq);
```

Either a single function is called, or a fitting function based upon a predicate is called from a list of candidates. Each parser function contains a predicate, that is used to determine if the corresponding function should be called, and the function itself:

```
1   typedef struct {
2       function ActionValue#(ParserStatus) _(pktType pkt) fun;
3       function Bool _() predicate;
4   } ParseFunction#(type pktType);
```

Each function can control the next stages of the packet parsing process using the `ParserStatus` return type:

```
1   typedef enum {
2       PARSE,
3       DROP,
4       OUTPUT
5       } ParserStatus deriving(Bits, Eq, FShow);
```

Each function can execute whatever functionality they like, as long as they are callable when the corresponding packet beat arrives. If the function is not callable, a fall back is called which. Three behaviors can be selected for this: (1) Drop the packet if processing can not keep

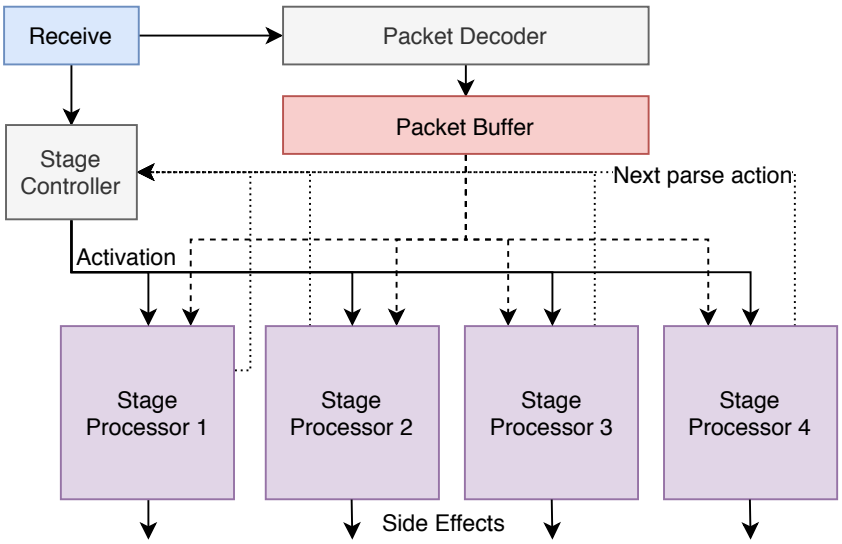


Figure 7.1: Overview of the system generated by the packet parser. Incoming packets are stored in a packet buffer that is accessible by the individual stages. The stage execution is controlled by the stage controller. As the stages are provided by the calling module, they might have side effects on the calling module such as writing or reading registers.

up, (2) continue processing but issue a debug warning, useful for simulation, and (3) block the pipeline and ensure processing.

Lastly, the user can decide if the Ethernet frame is buffered or not. If it is buffered then all stages can access the packet data of the preceding packets. Otherwise, the user has to make sure to save all critical data, ensuring very efficient memory usage.

The architecture that is generated by the packet parser is illustrated in Figure 7.1. The received part of the packet is forwarded to the packet decoder and the stage controller. The stage controller determines which stage should be active based on the individual predicates and the stage of the decoding process. The packet decoder takes the packet and extracts the bits into the desired packet format. This decoder can, for instance, convert the network byte order to little endian. The resulting data is stored in a packet buffer. The packet buffer can be circumvented if desired to reduce the memory requirements. The activated stages can access the packet buffer or the data directly, depending on the mode, and process the information as desired.

As the stages are provided by the user in the `PktParserStages` list, they can have side effects on the calling module. For instance, they can access data of the calling module or write data to registers. Each stage determines what should happen in the next stages. For example, the stage can enforce that the rest of the packet is dropped.

This is all that is needed to efficiently parse network packets. Parsing a packet that consists only of an Ethernet header requires only a few lines of code as shown in Listing 7.1.

The example code determines if the FPGA is the receiver of the packet based on the MAC address and prints out the Ethernet type in the second stage. Afterwards, the rest of the packet is dropped. Additional syntactic sugar is available to simplify some of the lengthy Bluespec syntax. The second stage could also be defined as:

```
parserStages = addSimpleFun(1, dropPacketTypes, True,
↪  parserStages);
```

or both stages can be added using

```
let parserStages = addSimpleFun(0, dropInvalidMac, True,
↪  addSimpleFun(1, dropPacketTypes, True, Nil));
```

Of course, this functionality is implementable in traditional HDL. However, the features added by Bluespec enable more focused work. The user can focus on processing the right data at the right time and does not need to know how the data arrives there.

This parser, and the similarly implemented packet creator, are the basis of the work presented in this chapter.

7.2 RELATED WORK

INPs is a topic of high interest in the research communities right now. Accordingly, many different approaches and research directions emerged. Acceleration of consensus protocols [72] such as Paxos [73] promises very low latency and high availability, making approaches feasible where highly fault tolerant main memory is moved into the network. Two main lines of research can be distinguished. The first approach focuses on enforcing particular behaviors in the network. By enforcing behaviors they can make strong assumptions about the network which can be used to decrease latency and increase throughput. These approaches have to fall back to traditional consensus protocols, if their assumptions fail. For instance, speculative Paxos assumes that messages are delivered in order. If this assumption fails, the protocol falls back to a reconciliation protocol.

Examples of this line of research include [78, 97]. The second approach including [26, 59, 62] move consensus algorithms directly into the network. Most of these approaches use some kind of programmable switch, usually based on the Tofino architecture [89], which imposes certain limitations.

For example in [26] the algorithm is only implemented for a *single* word of memory, because currently existing hardware in-network processing hardware is memory limited. FPGAs however do not impose such limitations and a different approach is shown in Chapter 9.

```
// Define the packet type.
// This packet consists of only the Ethernet header.
typedef struct  {
    Bit#(16) eth_type;
    Bit#(48) mac_src;
    Bit#(48) mac_dst;
} ThePacket deriving(Bits, Eq, FShow);

List#(PktParserStages#(ThePacket)) parserStages = Nil;
// First stage: Check if I am the destination
function ActionValue#(ParserStatus) dropInvalidMac(ThePacket pkt);
actionvalue
    let dst_mac = toggleEndianess(pkt.mac_dst);
    if(dst_mac != my_mac) begin
        return DROP;
    end else
        return PARSE;
endactionvalue
endfunction
// Add stage one to the stage list
// Will always be called on every packet
parserStages =
    List::cons(
        PktParserStages
            {stage: 0,
            fun: tagged FunctionSimple
            ParseFunction {fun: dropInvalidMac, predicate: True}},
        parserStages);

// Second stage: Retrieve the ethernet type from the packet
// Print a debug message and drop the rest of the packet
function ActionValue#(ParserStatus) dropPacketTypes(ThePacket pkt);
actionvalue
    let ethType = unpack(toggleEndianess(pack(pkt.eth_type)));
    $display("Received packet of type %x for me.", ethType);
    return DROP;
endactionvalue
endfunction
// Add stage two to the stage list
// Will always be called on every packet
parserStages =
    List::cons(
        PktParserStages
        {stage: 1,
        fun: tagged FunctionSimple
        ParseFunction {fun: dropPacketTypes, predicate: True}},
    parserStages);

// Generate packet parser based on the stage list
PktParser#(ThePacket) receiver <- mkPktParser(parserStages);
```

Listing 7.1: Parsing an Ethernet packet with the packet parser presented in Figure 7.1. The parser consists of two stages: The destination of the packet is checked and the packet is dropped if the receiver is not the destination. If the packet is valid, the Ethernet type is printed.

Database acceleration is another topic of high interest right now. For example [64, 119] move key-value caches into the network to increase bandwidth and reduce latency. Another topic of interest is moving certain primitives of distributed databases into the network. Chapter 8 presents an implementation of a hash join [32] inside the network, leading to a lower bandwidth requirement and faster processing times. Miao et al. show in [83] how a single switch can replace a large number of software based Layer 4 load balancers. An interesting approach is followed in [86] where an FPGA is used as a honeypot for attack tracing. As the wide range of different examples show, INP is a research question with a lot of appeal. Most of the current examples of INP are using Tofino-based software-defined switches for their high throughput characteristics. However, these switches are not suited for all possible applications. Especially stateful processing receives only a very limited support.

8

IN-NETWORK HASH JOIN

The work presented in this chapter has been partially published in:

1. Jaco Hofmann, Lasse Thostrup, Tobias Ziegler, Carsten Binnig, and Andreas Koch. “High-Performance In-Network Data Processing.” In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2019, Los Angeles, United States.* 2019

"Scalable database systems for analytical workloads such as Terradata, Microsoft Parallel Data Warehouse, or Amazon's Redshift are being used today for analyzing massive amounts of data in distributed setups." [54] Data is distributed across many nodes of a cluster to exploit the available processing resources. Operations on such data bases require certain movements of the data across the nodes to process the final result in a process called shuffling. The high communication and inefficient communication schemes cost associated with shuffling limits the speedup potential of such approaches, especially in main memory databases [1, 101].

High-speed networks and RDMA [131, 132] are one approach trying to solve the underutilization of the networks. However, apart from these approaches, that better utilize the available network bandwidth and promise lower latencies, newer research moves the calculation inside the switch instead. Through programmable switches, as introduced in Chapter 7, or FPGA, the calculations traditionally located in the individual nodes, can be computed inside the network itself.

This approach is already successfully applied to many problems that work without keeping large amounts of state, such as distributed SQL operations or in-network caching [15, 36, 103]. Nonetheless, stateful operations are difficult to realize in such a system with the current generation of hardware. The switches are either not capable of keeping up with line-rate or simply can not store enough state for operations such as SQL joins or aggregations [15, 77].

Accordingly, a new generation of switches for analytical workloads in the network requires different design decisions than the current generation of throughput optimized switches. The following pages present a different switch architecture for in-network analytical SQL workloads. The FPGA used for this tasks enables full design space freedom and exploration of novel architectures. The resulting architecture leverages the larger amount of off-chip-memory available to the FPGA to perform a stateful operation directly inside the network. The architecture is not limited to simple query execution but can execute

pipelines of multiple chained stateful SQL operators, for example hash-table building and probing.

The proposed architecture is tested in a state-of-the-art databases application for distributed hash-join. The query optimizer selects the desired operators to be run on the FPGA and loads the bitstreams with the desired functionality. Additionally, the query optimizer selects the best execution strategy for the given task [54].

Evaluation shows that the proposed switch architecture significantly speeds-up the distributed join processing by up to 7×, in a skewed shuffle scenario using a traditional approach. Typical ingress problems can be avoided by the in-network approach by reducing the communication between nodes.

Firstly, in Section 8.1 the architecture and integration of the proposed INP system is introduced. Changes to the query processing and optimization optimization stages to support the INP are described in Section 8.3. Furthermore, the section sepecifies details of the proposed architecture. Lastly, the evaluation, running a hash-join application in the network, is presented in Section 8.4.

8.1 BACKGROUND

Before delving into the architectural details of the demonstrator, the following section introduces the INP-based query processing scheme and includes a comparison to traditional distributed query processing.

Classical distributed query execution in a shared-nothing database consists of one master and several compute nodes. The system is connected through a central switch [54]. The execution plan in Figure 8.1 shows a typical execution plan for such a system, executed as the result of a SQL statement such as `SELECT * FROM A JOIN B JOIN C`. The join is implemented as a hash join which first has to generate hash tables for the different `B` and `C` tables before probing the join key of the `A` table based in the generated hash tables. Of course, a real world distributed database system has to go through some optimization steps to better utilize the available resource. The join could be implemented in a different way, but in this scenario the assumption that `B` and `C` are much smaller than `A` results in a hash join being the most optimal join to chose.

The classical execution begins by shuffling `A` and `B` according to the join key. Shuffling distributes the relation in question across the compute nodes to ensure even distribution of the data. After shuffling the workload of each cluster should be even and data belonging to the same join key should reside on the same node. Afterwards each node generates a hash table for their part of table `B`. Lastly, the nodes have to probe into the hash table with the data from `A` to generate $A \bowtie B$.

The process basically repeats for the second join. $A \bowtie B$ and `C` are shuffled, a hash table is generated from `C`, and finally, The hash table

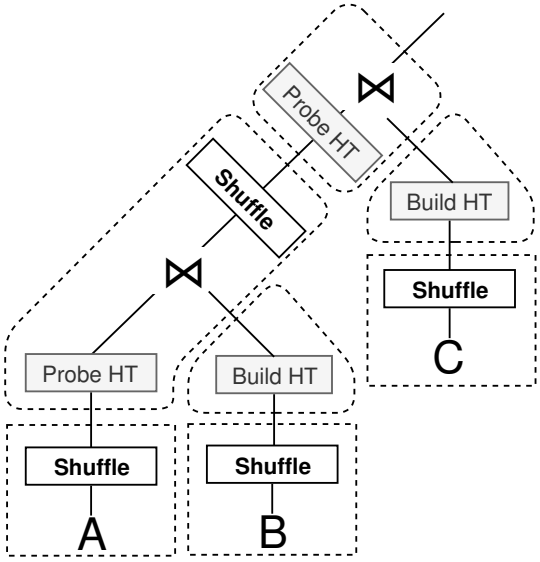


Figure 8.1: Example of Query Plan for Classical Execution from [54]. The plan executes `SELECT * FROM A JOIN B JOIN C` using two hash joins. Firstly, after shuffling of `B`, a hash table is generated. The hash table is probed with the shuffled `A`. Afterwards the same is done for `C` and the intermediate join result $A \bowtie B$.

of `C` is probed with $A \bowtie B$ to generate $A \bowtie B \bowtie C$. Accordingly, each join requires two expensive shuffling operations per join [54]. The shuffling mechanism, which is based upon the join key, might come with other unwanted consequences. If the join keys are not equally distributed it can happen that one node receives much more data than the other nodes. For one that single node has to process much more data compared to the other nodes. Furthermore, the ingress of the network connection of that node will be overloaded with the data coming in from multiple other nodes at the same time. Such a skewed scenario can completely eliminate many advantages of a distributed database.

This is where the scheme proposed hereafter improves upon the classical scheme. Instead of shuffling the data across the network several times, the network itself can do the calculations. The basic steps of the execution scheme are shown in Figure 8.2. A master nodes receives a new SQL query and compiles an execution plan based on it (Figure 8.2 ①). The master is responsible to distribute that plan across the nodes and configures the switch accordingly (Figure 8.2 ②). Afterwards the system is ready to process the query and all components begin execution.

The same request as executed in the classical scheme, presented in Figure 8.1, is also executed for the proposed scheme (See Figure 8.3). For the INP scheme the different types of operations, such as building a hash table or probing one, are called pipelines. The master node splits the request into separate pipelines and programs the individual

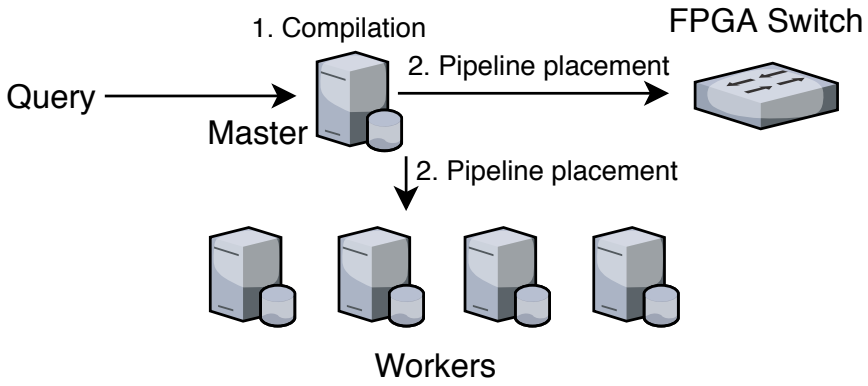


Figure 8.2: Overview of the system to process the INP scheme proposed in [54]. A master node is responsible to configure the other components of the system according to the incoming query. The system consists of multiple nodes that store relations and can do processing. Furthermore, a FPGA based switch is used to do calculations inside the network.

components. In this case, the switch is programmed to generate hash tables for `B` and `C` and probe those tables when receiving data from `A`. The nodes will send `B` and `C` to the switch which builds the corresponding hash tables. Afterwards, `A` is streamed through the switch. In the end the switch can directly compute $A \bowtie B \bowtie C$ based on the incoming data from `A`. Any shuffle operations that where previously necessary in the classical scheme are gone. Each node simply sends out the data it possesses without any concern for data distribution.

Accordingly, the main conceptual difference between the INP scheme and the classical one, is the elimination of shuffling, as well as, the elimination of the intermediate shuffling of join results. The switch can calculate all necessary hash tables simultaneously. The new approach avoids many of the problems associated with shuffling [54].

For instance, shuffling operations hinder parallelism of the execution. The following phases of the pipeline are blocked until the result of the previous step has been computed. In the example this occurs before the second join, which can not be processed before the shuffling of first join has finished.

Additionally, the shuffling results in very high network load as the result of every step has to be distributed across the network. Otherwise, each node lacks the necessary data to continue. In data warehouse scenarios this cost becomes prohibitively expensive. In such a scenario a large fact table is joined with multiple smaller dimension tables. The fact table, being orders of magnitudes bigger than the dimension tables, still has to be shuffled to all nodes. The shuffling process of the intermediate join results and the fact table dominates overall query execution cost [54].

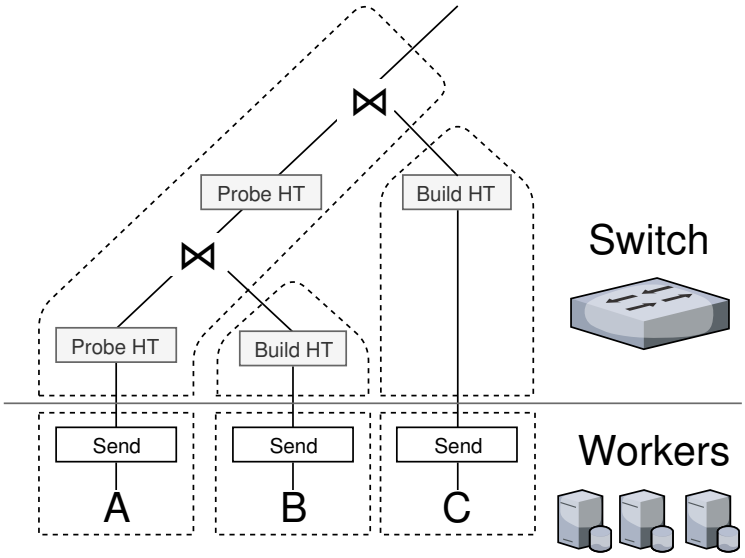


Figure 8.3: Execution of `SELECT * FROM A JOIN B JOIN C` in the INP scheme shown in [54]. The shuffling necessary in Figure 8.1 is gone as the switch can directly process incoming data. Furthermore, there is no intermediate join result anymore as the switch can generate and probe both hash tables simultaneously.

Lastly, the INP scheme is not susceptible to skew problems by avoiding shuffling operations altogether.

8.2 DESIGN

The proposed scheme comes with many changes to established database systems, which are described in the following section.

First of all, the query compilation has to be changed according to the scheme proposed in [54]. The master has to decide where to execute the different pipelines. The classical approach had only the nodes at disposal for computations, but for the INP approach, the FPGA switch has to be considered as well. As documented in Chapter 2 it is not feasible to generate a bitstream based on a single query. The synthesis and place&route times would be orders of magnitude larger than even long running queries in a classical system. Instead, the master has access to a set of pre-generated bitstreams for different kinds of pipelines. The master can reconfigure the switch based on the desired functionality in a couple of milliseconds. The system can furthermore fall back to traditional execution if the given pipeline is not supported by the switch or the master determines using the FPGA as slower.

The bitstreams come with different kinds of parametrizability. For instance, one bitstream is highly optimized for hash table generation of key-value-pairs with 4 B of key and 4 B of value. Another one might support any size between 1 B and 32 B for either data. Processing a (4 B, 4 B) tuple with the flexible architecture is possible, but results in

lowered performance of the switch. Accordingly, the master choses the optimal signature based upon the characteristics of the query to avoid unnecessary steps. In particular, consider that `B` of Figure 8.3 is using a (4 B, 10 B) setup and `C` is (8 B, 10 B) configuration. The master could not use the specialized architecture for (4 B, 4 B) but has to fall back to the generalized architecture avoiding as much overhead as possible. At the same time, this case should not be common considering that most workloads are known beforehand and specialized architectures can be created for them.

Similarly, the optimizer, a system used to find the best execution plan, has to change. Considering the change in the execution model, the optimization objects of the existing cost-based optimization need adjusting [54]:

- Avoid reshuffling by processing most of the operations in the switch. Avoiding pipeline-breaking operations improves overall performance in such a system.
- Consider the characteristics of the switch. An FPGA with 8 GB of memory can not create a hash table for a table with 16 GB of data. The optimizer might move only certain operations into the switch or process queries in multiple steps.

The feasibility of the INP approach is shown in the scenario already introduced in Figure 8.3. These "left-deep join trees with primary-foreign relations" [54] are commonly found in analytical workloads performed on databases and provide optimal opportunity to show the effect of shuffling avoidance.

Initially, before starting with the implementation, the theoretical benefits of such a model should be considered. Consider a left-deep plan consisting of a left-deepest relation L, where L can be an input relation or an intermediate result. Furthermore, a set of tables T_i which are to be joined with L, analogous to the query plan in Figure 8.3. The number of workers, the nodes previously introduced, is defined by N. $|R|$ denotes the cardinality of any relation R and $ts(R)$ the size of the tuple in R [54].

The optimizer determines a subset of tables indexed by I for INP as defined as $L \bigcup_{i=1}^{I} T_i$ [54].

Going back to the example: The complete operation has to join all three tables A, B, C. Nonetheless, the optimizer might determine that joining A and C is feasible on the switch, but B is too large. Hence, after processing $A \bowtie C$ inside of the switch, the join between the intermediate result and B occurs traditionally using shuffling.

The following cost models for executing `SELECT * FROM A JOIN B JOIN C` in either system are based on these notations.

COST MODEL FOR CLASSICAL MODEL Before calculating the cost for processing the whole query, the cost for moving a single relation

across the network has to be determined. The corresponding cost value c can be calculated as described in [54]:

$$c_{rel}(R) = |R| * ts(R) \tag{8.1}$$

The cost of moving a single relation across the network can be combined to calculate the network cost of joining the relations indexed as I:

$$\begin{aligned} c_{shuffle}(I) = \frac{N-1}{N}\Big(&c_{rel}(A) + c_{rel}(A \bowtie T_{i1}) + \dots \\ &+ c_{rel}(A \bowtie T_{i1} \bowtie \dots \bowtie T_{im-1}) + \sum_{i \in I} c_{rel}(T_i)\Big) \end{aligned} \tag{8.2}$$

In [54] the example shown in Table 8.1 is used to explain the formulas. The example uses a scenario with four works ($N = 4$).

Table 8.1: Example parameters for a two way join as shown in Figure 8.3. Taken from [54].

Relation	Size (|R|)	Tuple Size (ts(R))	$c_{rel}(R)$
A	100 000	32	3 200 000
B	10 000	10	100 000
C	10 000	10	100 000
AB	100 000	32	3 200 000

The example considers the complete plan so that I consists of A, B, C. Accordingly, the costs associated with each relation have to be entered in the equation as shown in Equation (8.3). Additionally, the intermediate result $A \bowtie B$ has to be shuffled which adds additional cost of 3 200 000.

$$c_{shuffle}(I) = \frac{3}{4}(3\,200\,000 + 3\,200\,000 + 100\,000 + 100\,000) \tag{8.3}$$

The example so far calculates the cost for moving all of the tables across the network. However, usually a certain percentage of each relations can remain on the individual node. Consequently, not considering skew $\frac{3}{4}$ of the data has to be shuffled and $\frac{1}{4}$ of the data can remain on the worker. In the end the cost of both joins in this scenario is $\frac{3}{4} * 6\,600\,000 = 4\,950\,000$ [54].

COST MODEL FOR INP In like fashion, the cost for the INP scheme is calculated, largely simplifying the cost equation by removing the shuffling cost leaving only the raw cost for moving the relations over the network once. Consequently the cost for the INP approach depends solely on the size of the relation and the size of the tuples:

$$c_{INP}(I) = \left(c_{rel}(L) + \sum_{i \in I} c_{rel}(T_i))\right) \tag{8.4}$$

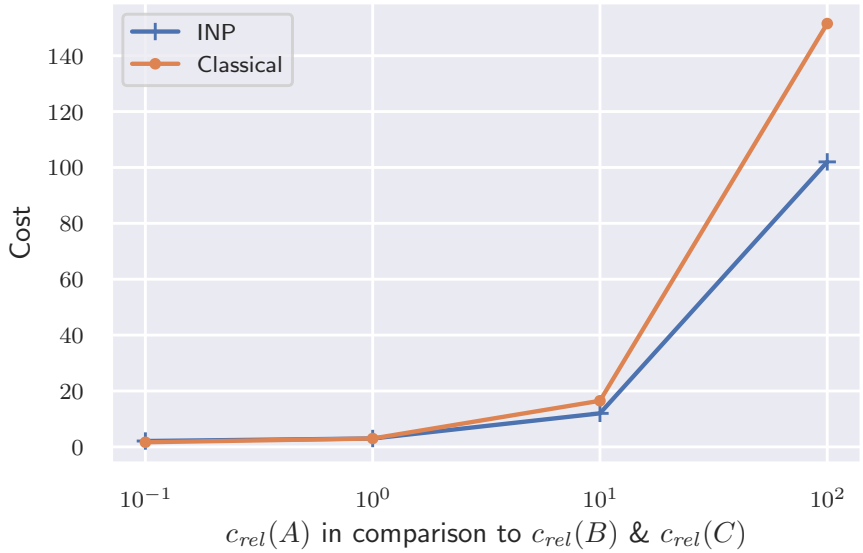


Figure 8.4: Cost for executing the traditional and classical schemes with different relative sizes of A compared to B and C. For very favorable scenarios, where the cost of A is small compared to the other relations, the classical approach is slightly better than the INP approach. However, for large relations A the INP approach wins out. (Taken from [54])

Applying the new equation to the example introduced in [54] results in

$$c_{INP}(I) = (3\,200\,000 + 100\,000 + 100\,000) = 3\,400\,000 \tag{8.5}$$

As a result, the INP approach is $\frac{4\,950\,000}{3\,400\,000} = 1.455$ times faster than the classical approach in this scenario. In general the absolute decrease in cost can be written down as $c_{INP}(I) - c_{shuffle}(I)$. The task of the optimizer is finding an optimal set I for which the cost is minimal but the memory constraints of the switch $\sum_{i \in I} |T_i| \leqslant C_{RAM}$ are met. Additionally, $I = \emptyset$ is valid, which means that INP is not applied at all [54].

The cost model can be used to analyze the behavior of the two schemes for different relations. This analysis can then be used to determine when INP is beneficial.

Equation (8.1) shows that the size (number and size of tuples) of relation A has major impact on the cost of either approach. For this reason, Figure 8.4 shows the effect of increasing the size of relation A in relation to B and C. The scenario still uses four workers $N = 4$ [54]. The relative size of A compared to B and C is plotted along the X-axis. The corresponding cost $c_{shuffle}(A, B, C)$ and $c_{INP}(A, B, C)$ is plotted along the Y-axis.

The classical approach wins out for small costs of A compared to B and C, because the cost of shuffling only part of the relation is cheaper

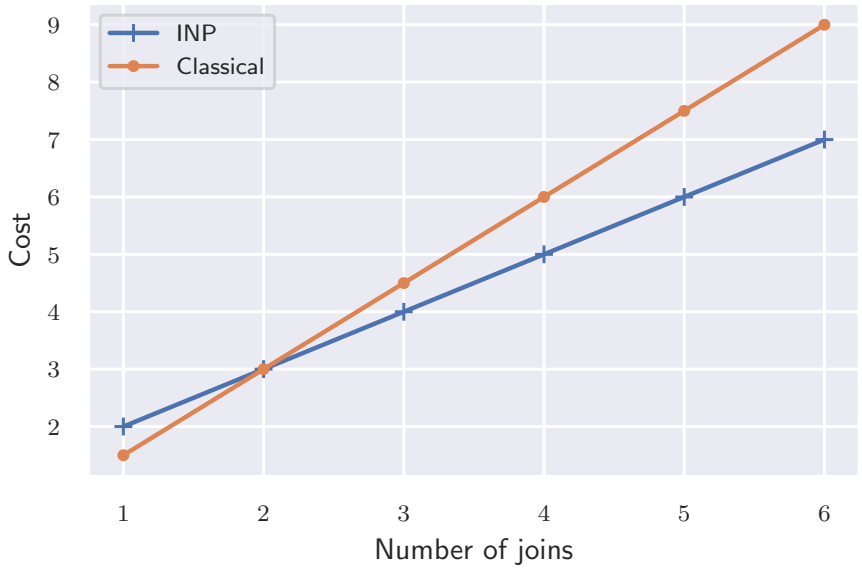


Figure 8.5: Cost Analysis for varying number of joins. Relation costs (c_{rel}) are kept the same for all joined relations.

than moving all the data to the switch. Nevertheless, for larger costs of A the shuffling cost increases rapidly and the INP approach wins by avoiding expensive reshuffling of the intermediate results [54].

Apart from the relative sizes of the relations, the number of joins influences the decision for either approach [54]. The plot in Figure 8.5 illustrates the cost associated with executing an increasing number of joins for relations at the same cost. The graph shows that the classical approach is superior for one join, and tied for two joins. However, for more joins the INP scheme pulls ahead. This is explained by the high number of intermediate joins required by the classical approach. The INP approach can, given sufficient memory, avoid all of these relations.

These numbers show that the INP approach should be considered for large A relations compared to B and C and if the number of joins is high. Either scenario, and often a combination of both, are common in data warehouse scenarios [54].

8.3 IMPLEMENTATION

The INP scheme relies on a central switch that is capable of executing pipelines. The prototype architecture described here can be used to offload hash table generation and probing.

The proposed switch is built upon the network parser described in Section 7.1. Accordingly, the architecture is written completely in Bluespec and integrated into TaPaSCo using the plugin presented in Section 5.7.

However, the switch does not act as a standard TaPaSCo PE in this case. A PE in the TaPaSCo sense executes a single operation and then terminates, sending an interrupt to the host. The switch on the other hand runs permanently and does never terminate. Nonetheless, TaPaSCo provides a high level of abstraction that can be used, despite not using the intended workflow. The TaPaSCo API is used to program the PE and read status and performance information from the core. Starting the PE has the effect of executing the desired command, for instance, setting the start address and length of a hash table. After executing the command, the PE signals completion to the host as intended. In addition, TaPaSCo simplifies porting the switch over to another platform.

For now the implementation utilizes the SFP+ plugin of TaPaSCo in conjunction with a NetFPGA SUME [133] FPGA development board. The board comes with a Xilinx Virtex 7 FPGA and 8 GB of memory across two DDR3-SDRAM DIMM modules. Furthermore, the NetFPGA SUME is intended for network targeted FPGA architecture design and comes with four SFP+ ports to connect the board to 10 Gbit/s Ethernet. As usual, the SFP+ ports support either Direct-Attached-Copper (DAC) or fiber optic cables.

The demonstrator is designed to perform a three way hash join in the network. Hence, the switch has to support generating and probing three hash tables simultaneously while serving four SFP+ ports running at line rate. The SFP+ ports are abstracted as 64 bit wide AXI-4 Stream interface running at 156.25 MHz each.

The architecture is divided into three distinct parts that are described in detail later in this section:

- Parse incoming packets. Discard any packets of no interest. Extract the operation from the packet and collect the tuples. Forward those tuples to the correct processing unit.
- Hash the requests that are intended for hash table generation.
- Probe the requests that are intended for probing.

The simplest task is performed by the parsing stage. As each of the units is independent of one another, they can all run at a relatively slow 156.25 MHz. The packet parsers, as illustrated in Figure 8.6, generate a stream of output tuples for hash table generation, or values to probe for probing requests. Parsing is done using the default configuration of Section 7.1. The only performance requirement that is that the parsing stages can never block. Accordingly, the units have to make sure that any tuple that can currently not be forwarded to the later stages is either stored in a temporary buffer or dropped. Stalling the network interface would lead to synchronization problems which would result in even more problems down the line.

Moving the tuples to the next stages is a more delicate endeavor. As the parser stages can never stall, efficiently forwarding the data to the

corresponding units is of paramount importance. Each parser stage has one large FIFO buffer towards each of the downstream units to store incoming tuples. These buffers are realized using one BRAM cell each and can store up to 1024 tuples.

The buffers are connected to an arbiter that is responsible for selecting the next tuple to hash or probe. Each of the downstream units has their own arbiter infrastructure. The arbiters run in a round-robin mode without fixation.

Currently, the architecture utilizes three hash units which can be fed in parallel to simultaneously generate three hash tables. Accordingly, the best performance is achieved if the three relations intended for probing are transmitted interleaved. Moving one relation after another would effectively remove the opportunity for parallelization of this step. Using multiple units to generate the same hash table would be detrimental for the overall performance as the infrastructure to avoid collisions results in a long critical path. Furthermore, this approach would decrease memory controller performance.

The probing infrastructure comes with three units, as well. However, each probing request has to probe into all the generated hash tables which means that probing multiple requests in parallel is not possible. As probing is easier than table generation, the overall system performance should not be impacted by this limitation.

The hashing infrastructure has to meet very tight performance requirements to keep up with the four times 10 GB of traffic. The maximum traffic can be calculated based on the border conditions of the system. Assuming each interface can transfer a theoretical maximum of 1.25 GB/s. Each Ethernet frame has a maximum size of 1524 B in this application, resulting in 820 209 $packets/s$. Removing the headers from the full frame, leaves 1468 B for tuple requests. As each tuple requires 32 B, this results in 45 $tuple$ per packet. Hence, $\frac{820\,209\,packet/s}{45\,tuple/packet} = 37\,627\,087\,tuple/s$.

The peak inserts per second that the infrastructure has to support is 37 627 087 per SFP+ interface. The whole design has to focus on very high throughput. Latency is less of an issue as feedback to the sender, such as an ACK mechanism, is currently not implemented. The hashing units can focus completely on inserts as delete operations are not necessary for the use case.

Hash table generation on FPGA is not a new thing. Research is especially prevalent for key-value-stores on FPGA [120, 123]. Executing hash table generation on FPGA is a completely different beast than performing that task on CPU. Efficient implementations for CPU have to ensure that the caches are employed properly. FPGA, however, do not have fast caches but come with different advantages. The CPU is limited to process small words, usually less or equal to 64 bit. On the contrary, FPGA do not come with such limitations and can process much larger words. For instance, the FPGA can process the memory

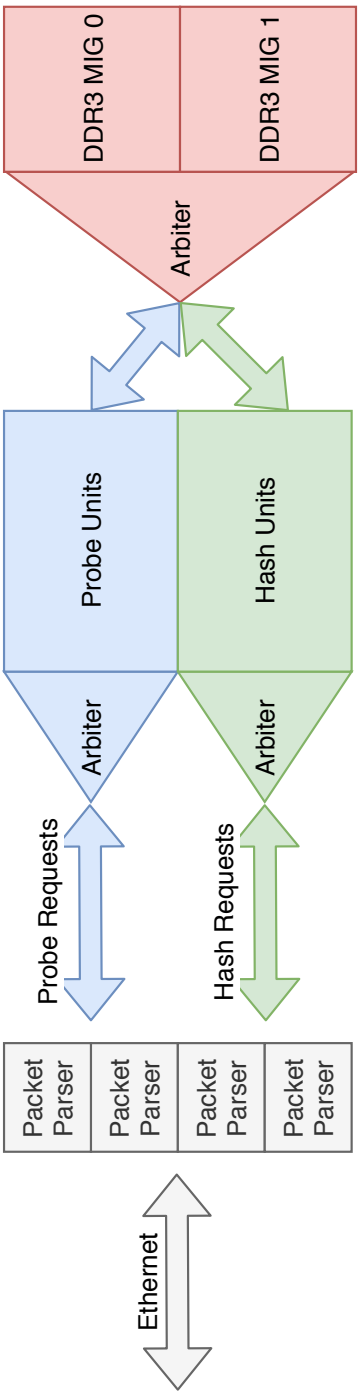


Figure 8.6: Overview of the proposed architecture on the NetFPGA SUME board. Data is processed as a stream of 64 bit words provided by the Xilinx 10G Ethernet Subsystem. The Ethernet packets are parsed using a Bluespec-generated packet parser. The extracted hashing and probing requests are forwarded to the hashing and probing infrastructure.

data width of 512 bit directly. Consequently, a different approach has to be taken and directly porting a CPU focused algorithm is unrewarding.

As previously mentioned the design has to be throughput optimized. The slowest component in the system are the two memory controllers. Hash table generation requires a lot of random access reads and writes of the memory and will be limited by this performance metric. Accordingly, the design has to ensure that the requests are as optimal as possible for the characteristics of the memory. For DDR3, for example, the number of bank machines has to be considered and requests have to be spread across the available resources.

The design utilizes buckets of 512 bit width, which corresponds to the width of the memory interfaces. Accordingly, fetching or writing a single bucket requires only one read or write operation. The units is fully pipelined and can execute the different stages in parallel to one-another. The basic flow is as follows:

1. Hash the key provided in the request by the parsing stage to calculate a bucket
2. Retrieve the corresponding bucket from the main memory
3. Place the key and value tuple in the first free position in the bucket
4. Write the bucket back to main memory

The architecture is susceptible for read-after-write hazards. If a bucket is requested a second time before the first request is written back to the memory, the following write would overwrite the data previously written. This problem can be solved by using look-ahead buffers at the cost of a slightly longer critical path. The look-ahead buffer contains previously requested data and provides this data to secondary requests if necessary. As the architecture is limited by the memory controller performance, anyway, this approach is chosen to ensure data correctness. The hash unit is illustrated in Figure 8.7.

The hash units store the hash tables interleaved (See Figure 8.8). This means that each hash table is not stored in a block of data. Instead, the first address stores the first bucket of the first hash table, the second address the first bucket of the second hash table and so on. This ensures that the memory controllers are evenly hit even with all units active at the same time.

As the architecture is limited by the random access performance of the DDR3 controllers, the typical measured number of inserts per second is only 56 348 300 which is close to 100 % of the available bandwidth. Newer devices which come with different types of memories can utilize the same architecture for much higher performance. A good candidate for this purpose are the new HBM2 Xilinx FPGAs.

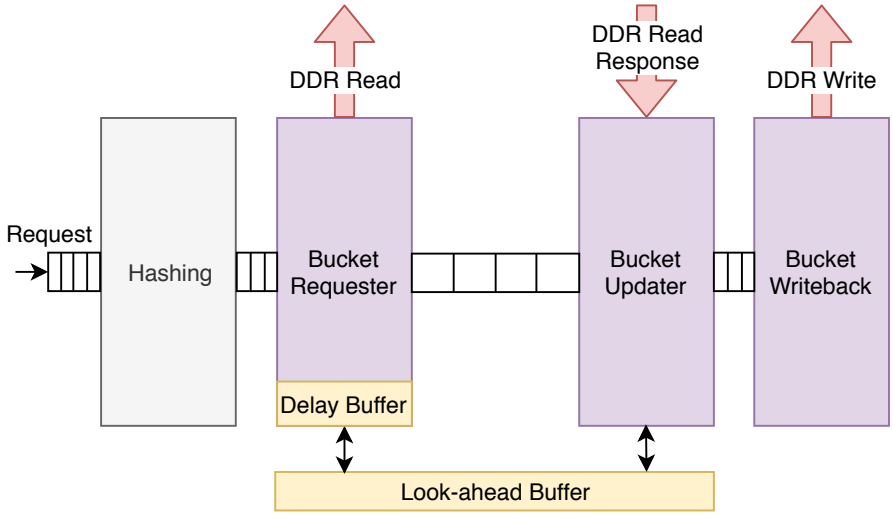


Figure 8.7: The hash unit is responsible for storing hash requests in the hash table. The first step is the hashing of the request to determine the bucket. Secondly, the corresponding bucket is requested from the external DDR memory. The bucket is then updated with the values from the request and written back to the memory. If a bucket is requested a second time before the first request has been answered, the value can be fetched directly from a look-ahead buffer. The second request is stored in a delay buffer until the previous request has been completed.

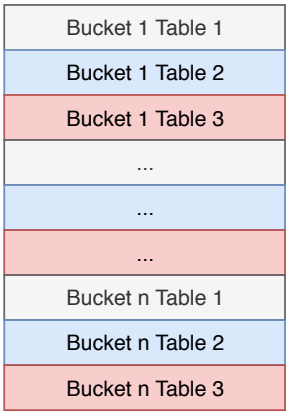


Figure 8.8: The hash tables are stored interleaved, instead of storing them in a block. This pattern helps spreading the load over both memories during hash table generation.

The performance of this architecture is completely determined by the random access Read/Write performance of the DDR3 controllers and is typically around 56 348 300 inserts per second per memory controller. Higher performance can be reached by utilizing newer devices with memories such as HBM having higher random access speed.

The probing units are simpler, but have to meet tighter performance requirements. The task of probing is difficult to parallelize across the units as each probing request over the network results in one probing request per hash table. Consequently, one 10 Gbit/s can generate about 119 836 254 requests per second.

On the other hand, the probing units do not need any writes, which frees up some memory bandwidth to be used for probing. The design looks similar to the hash units, as shown in Figure 8.9, lacking the last two stages, instead returning the probe result:

1. Hash the key provided in the request by the parsing stage to calculate a bucket
2. Retrieve the corresponding bucket from the main memory
3. Return the key contained in the bucket, or an invalid flag if the key is not found

The probe units are measured to serve about 75 % more requests at around 98 430 500 probes per second per memory controller. Once again, splitting the hash tables over both memories is very important to reach peak performance.

Whereas, the hash units do not produce any output, the probe units will produce the probe result. The result of all three probes is collected and forwarded to the requesting parsing unit. The parser is then responsible to generate a response packet and send it out via SFP+.

All in all, the slow DDR3 controllers of the NetFPGA SUME fail to meet the requirements to serve four SFP+ connections. The system in total is able to process around 29.9 Gbit/s of network traffic for the hashing site. The probe units can process approximately 196 861 000 tuples per second.

8.4 EVALUATION

The evaluation presented hereafter is only a small step towards the system envisioned. However, the numbers show that the theoretical advantages, in this case for a three-way hash join, can also be achieved in practice.

The evaluation is performed on five nodes powered by Intel Xeon Gold 5120 CPU @ 2.2 GHz [54]. Each node has access to 384 GB of memory and a 10 Gbit/s network interface, running Ubuntu 18.04.

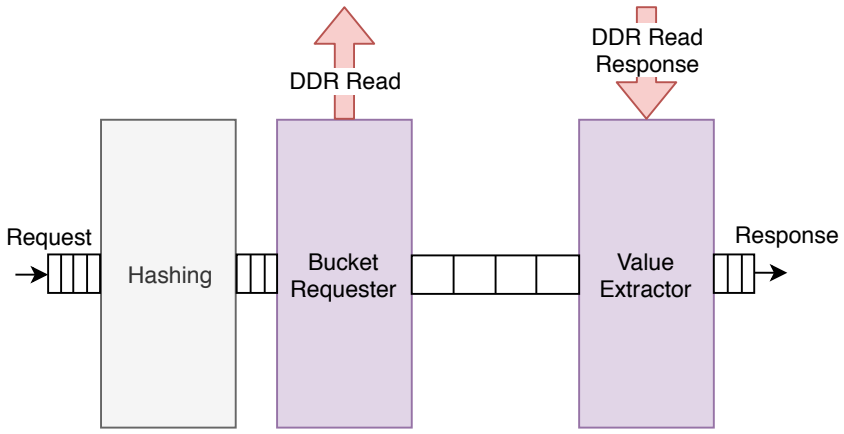


Figure 8.9: The probe unit retrieves values from the hash tables. The hashed key is used to receive the corresponding bucket. The value extractor checks if the bucket contains the value, which it returns in that case. Otherwise, it will return an error code as a response.

Four nodes are reserved for computation while one node acts as the master node as shown in Figure 8.2. Furthermore, a NetFPGA SUME board running the architecture presented in Section 8.3 serves as the FPGA switch. As the FPGA is using SFP+ switch connections, and the nodes are connected via RJ45, a Zyxel XS3700 performs the conversion between either standard. The RJ45 side uses CAT 6 cables and the SFP+ side uses two DAC cables from Digitus and two SFP+ fiber optic transceivers from FLEXOPTIC. Figure 8.10 illustrates the experimental setup with the FPGA highlighted in red inside a host PC that is only used to control and program the FPGA but does not perform any computations. The rack housing the nodes barely visible at the bottom. Finally, the XS3700 switch that converts the connections is shown in front.

The system, dubbed *NetJoin*, is able to perform HashJoin operations efficiently using either the classical approach or the new INP focused one. Accordingly, a shuffle-heavy scenario is chosen as described in Section 8.2. Just like in a data warehouse where a fact table with foreign keys are joined with the dimensions tables, the experiment joins a table `A` with three tables `B`, `C` and `D` [54].

The tables are pre-partitioned in away that ensures that the first join can not be done without reshuffling. In a real world scenario the data could be partitioned in a way that the first join works without reshuffling. Nonetheless, both processing schemes would benefit about equally of this optimization and it will not be considered here [54].

The prototype lacks certain features that are needed in a real world system. For example there is no mechanism to resend dropped data,

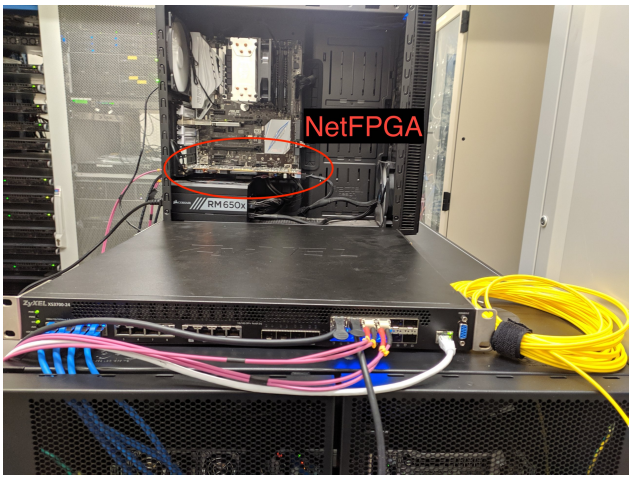


Figure 8.10: Experimental setup around the proposed NetFPGA SUME based FPGA switch. The server rack at the bottom houses four compute and one master node based on Intel Xeon 5120 CPUs. The central Zyxel XS3700 switch connects the experimental setup as the nodes come with RJ45 based network interfaces and the FPGA requires SFP+. Taken from [54].

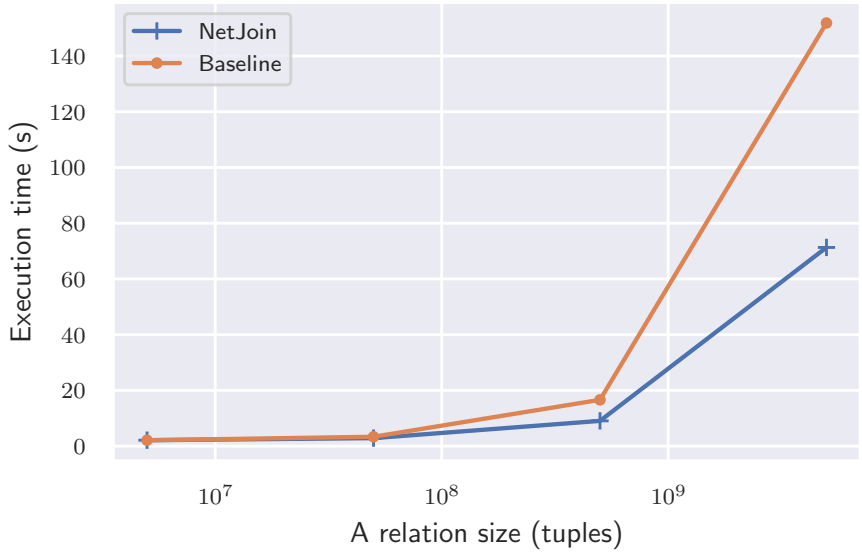
instead, the experiments ensure that at most 2 % of the tuples are dropped. This approach is common in other INP papers as well and ensures focus on the given scenario [77]. To ensure reliability, the protocol can be changed to employ higher layers such as TCP or use a lightweight protocol instead for RAW Ethernet frames. These changes only affect the parsing stages of the FPGA design and would not incur large performance penalties.

The platform is used to perform three different experiments to validate the theoretical calculations shown in Section 8.2.

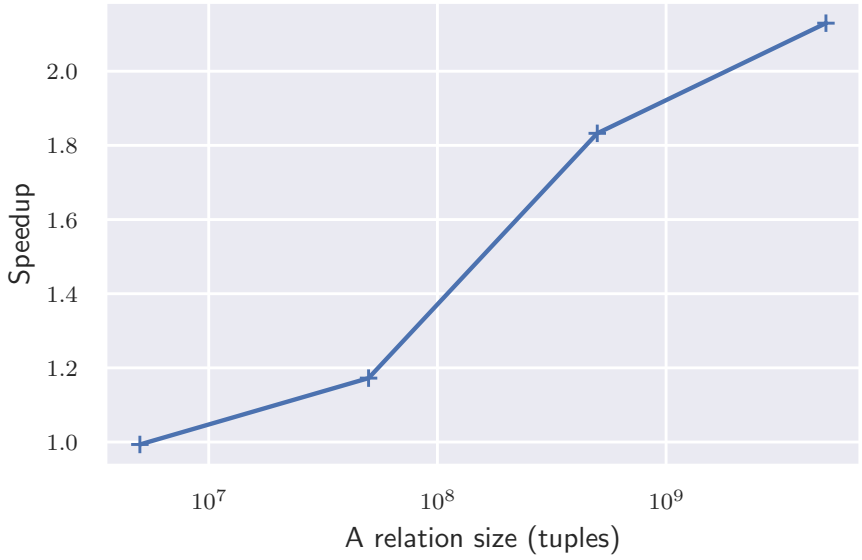
8.4.1 *Experiment with skew*

The first experiments replicates the calculations presented in Figure 8.4. The size of the `A` relation is increased relative to the sizes of the other three relations. The graph compares the runtime of the distributed hash join for either approach (Figure 8.11a) and indicates the speedup of the INP approach over the classical scheme (Figure 8.11b). The `B`, `C` & `D` relations contain 50 000 000 tuples each, while the `A` relation is scaled across 5 000 000 to 5 000 000 000 tuples. The experiment ensures that all nodes receive the same number of tuples by using uniformly distributed join keys.

The theoretical results are confirmed by the measurements. The classical approach rapidly loses out to the more efficient INP scheme which avoids relation shuffling. The scheme is only competitive when the `A` is smaller than the other relations. The overhead associated with shuffling is too big even though the nodes do no processing and have



(a) Join runtime.



(b) Speedup of *NetJoin* over baseline.

Figure 8.11: Experimental evaluation of the findings presented in Figure 8.4. Three relations are kept at 50 000 000 each, while the last relation is scaled from 5 000 000 to 5 000 000 000 tuples. The experiment is performed with four nodes each running at 5 Gbit/s. The numbers show that *NetJoin* quickly outperforms the classical approach. Only for few tuples in A the shuffling overhead is small enough to be competitive. The numbers have been adopted from [54].

Figure 8.12: Bandwidth required per node, in a four node configuration, for NetJoin to match the runtime of the baseline running at 5 Gbit/s.

to send out their complete data, compared to only $\frac{3}{4}$ of the data in the classical scheme. The speedup of the INP approach over the classical one increases with larger sizes of A. It reaches 2× for the largest evaluated configuration where the A relation is 100× bigger than the remaining relations. The difference in performance can be used for different purposes. For instance the necessary bandwidth to perform the join in a given time can be reduced. Similarly, the computation time can be decreased at the same bandwidth. Figure 8.12 shows the bandwidth per node necessary to match the baseline running at 5 Gbit/s. To achieve the same performance using a 5 000 000 000 tuple A relation only 2.347 Gbit/s are necessary. The saved bandwidth helps alleviate common data center issues such as ingress congestion.

Shuffling is especially expensive if the join key is not equally distributed. In the worst case, one node receives the majority of the tuples, while the other nodes idle. This is the worst case for a traditional system, but does not affect *NetJoin*. The experimental setup to show this behavior consists of a scenario where node 1 receives 80 % of the tuples, node 2 receives 13 %, node 3 5 % and the remaining 2 % go to node 4 [54].

As the protocol does not ensure data integrity, special care was given to the adequate throttling of the network. Otherwise, most of the packets targeted at node 1 would have been dropped by the switch as the ingress link of the node is congested.

8.4.2 *Experiment with skew*

The measurements in Figure 8.13 confirm the vulnerability of the traditional scheme to skew. The distributed join suffers from multiple

issues: (1) One node has to process 80 % of the data, overall runtime increases due to the lack of parallelism, (2) the reduced network bandwidth necessary to avoid dropped packets leads to slower data movement into node 1.

NetJoin is affected from neither issue and achieves the same performance as before. The skewed key does not have any influence without any shuffling taking place. Accordingly, Figure 8.13a illustrates that *NetJoin* shows identical runtime while the baseline is up to 3.31× slower in the skewed scenario. For this reason, the speedup increases to 7× for the largest `A` relation.

8.4.3 *Number of Joins*

To finish up the evaluation presented here, another experiment is performed in which the number of joins is varied while the size of the relations is kept constant. This experiment aims to show that the assumptions in Figure 8.5 are correct. Not only the relation size increases the overhead from shuffling, but also the number of joins that are performed as each intermediate result has to be shuffled. Accordingly, the relations sizes are all fixed to 50 000 000 tuples and the number of joins is varied from one to four. The results in Figure 8.14a confirms the theoretical considerations. The classical scheme is slightly faster processing one join. For two joins both approaches are almost equal. Afterwards, the runtime of the INP scheme grows slower and the traditional approach is increasingly slower. The same experiment can be repeated for varying sizes of `A` resulting in the same conclusion with *NetJoin* increasing the lead [54].

8.5 CONCLUSION

This chapter introduces a new type of distributed database system where shuffling of relations can be avoided by moving processing into a central switch. This work is published in [66]. The switch is controlled by a new query optimizer that can incorporate the switch to process pipelines. Off-the-shelf programmable switches are not up to this task as they lack the necessary amount of state. For this reason, an FPGA is the ideal demonstrator for this approach. The FPGA can be flexibly adopted to different tasks and can keep up with line-rate processing of stateful database operations.

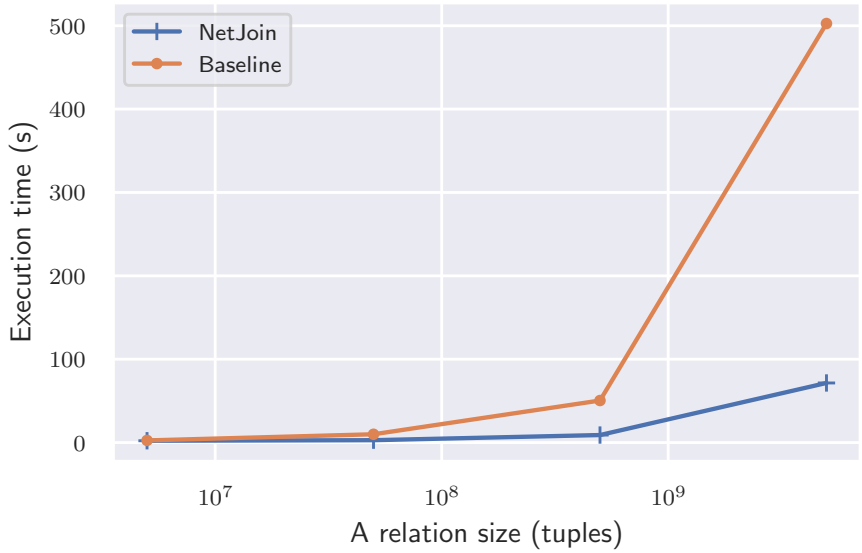
Furthermore, [54] describes the necessary changes to a database system to enable pipeline processing in the network. The optimizer and compilation stages aim to avoid shuffling of relations across the network by instead moving the processing into the switch.

The approach is evaluated for a four-way hash join. The evaluation shows that the traditional approach is inferior in well behaved sce-

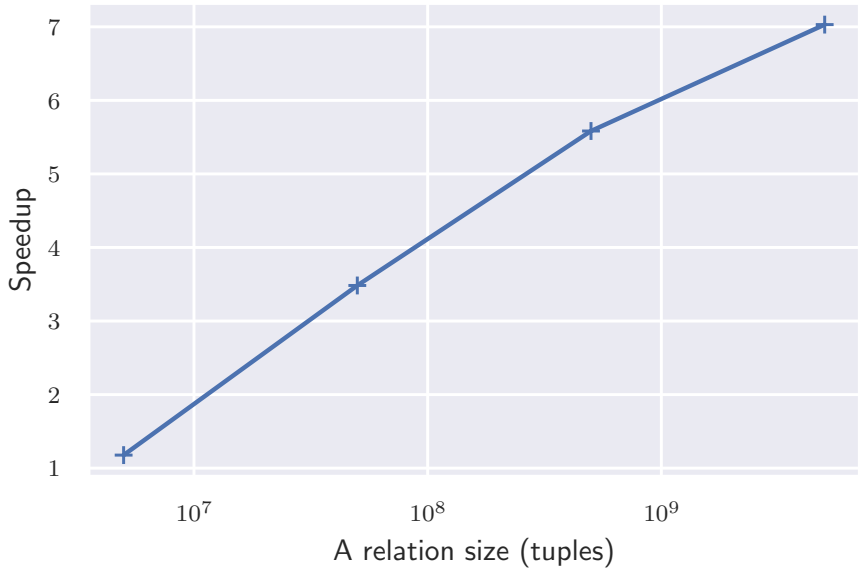
narios, and loses more ground in heavily skewed scenarios. The INP approach provides a speedup of up to 7× in this case.

As mentioned in multiple parts of this chapter, the prototype does not implement all the features proposed. These problems range from network related issues, for instance, the output of the switch can not be any bigger than the ingress due to congestion issues, to more hardware problems. The memory controllers of the FPGA used for the prototype are a bottleneck to the overall system performance. Newer generations of FPGA which employ HBM2 and large amounts of DDR4-SDRAM provide the necessary features to work towards 100 Gbit/s processing inside the network, by enabling more sophisticated caching schemes and raw performance [54].

Applying the proposed changes to the real world requires additional work in a variety of areas. Complex topologies with multiple switches are not explored, yet. These topologies offer more possibilities for parallelism but come with increased maintenance and control overhead. Lastly, fault-tolerance or isolation is not considered right now [54].

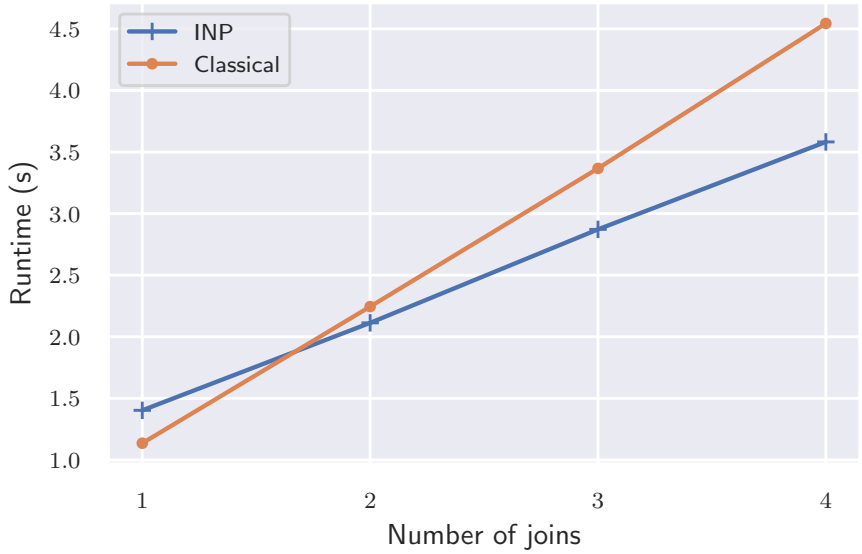


(a) Join runtime.

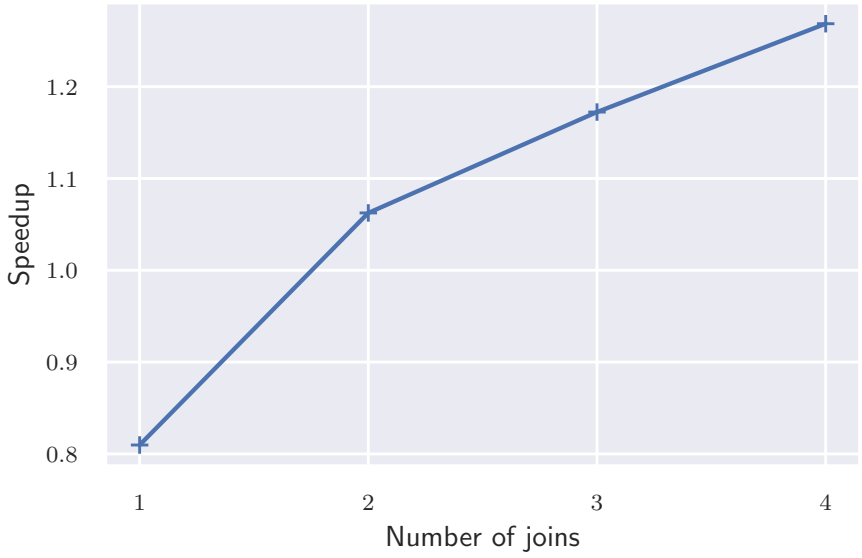


(b) Speedup of *NetJoin* over baseline.

Figure 8.13: The same parameters as chosen in Figure 8.11a applied to a heavily skewed join key scenario. Instead of equal distribution, node 1 receives the majority of the data, while the other nodes are underutilized. Accordingly, node 1 has to shoulder most of the work and the network ingress is overloaded. This shows in the results as the runtime of the join increases by up to 3.31×. *NetJoin* is not affected at all by these changes and performs equally well. Graphs adopted from [54].



(a) Join runtime.



(b) Speedup of *NetJoin* over baseline.

Figure 8.14: Experiment 3. Scaling number of executed joins in query from 1 to 4. All relation sizes are fixed to 50 000 000 tuples. *NetJoin* is slower for 1 join since the complete relations are sent to switch. With more joins *NetJoin* outperforms the baseline by avoiding shuffling intermediate joined relations. With relation `A` being bigger than joined relations, the speedup increases further as demonstrated in Experiment 1.

9 CONSENSUS IN THE NETWORK

The work presented in this chapter has been partially published in:

1. T. Dang, Jaco Hofmann, Y. Liu, M. Radi, D. Vucinic, and F. Pedone. "Consensus for Non-volatile Main Memory." In: *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. 2018

DISCLAIMER The results presented in this chapter were partially developed during an internship at Western Digital. Hence, certain details of the accelerator can not be discussed here in full depth. The findings presented are limited to those that have already been published.

INTRODUCTION Todays computers have different tiers of memory, from very fast SRAM caches inside each CPU core, to main memory in the form of DDR up to very large non volatile memories in the form of Hard Disks (HDD) and Solid State Disks (SSD). The memories have different characteristics, namely, their response time, volatility and cost [30]. The SRAM caches have very low latency but come at a very high price which makes them expensive. Hard Disks are very large, in the order of tens of terabytes, but also very slow. Whereas, the caches can be accessed in ns, the SSDs and HDDs require many ms to retrieve data. Accordingly, the tiered approach reduces latency for data which is currently being worked on and kept in the caches, while still providing large non-volatile storage options for other data. Additional problems with the write characteristics of SSDs lead to additional complexities, such as the limited write endurance which as to be alleviated through wear leveling.

The rise of Storage Class Memories (SCM), such as Phase-Change Memory (PCM) [124], Resistive RAM (ReRAM) [3], and Spin-Torque Magnetic RAM (STT-MRAM) [66], aim to shake up the common structure of tiered memory. These memories are non-volatile, offer byte-addressability and are, at the same time, not much slower than traditional DRAM. Furthermore, these memories are based on simple memory cells that allow for denser packing, leading to cheaper prices compared to DRAM.

Newer developments in physics [19] tip the scale further by increasing the memory utilization. Cross-point PCM that is already available as Intel Optane® M.2 is about 6.7x cheaper per gigabyte at retail compared to DRAM. The low access times and large, non-volatile storage provided through SCMs enables new memory hierarchies where

DRAM and HDDs/SSDs are replaced with a single tier of SCMs. At the cost of slightly higher access latencies compared to DRAM, the system would receive "a single, cost-effective and uniform type of memory" [30]. Additionally, the non-volatile nature of SCM could enable totally different types of applications compared to volatile DRAM, for example for databases.

The prospect of these techniques is appealing, but SCMs come with certain caveats. As all known SCM technologies are based on atom movement, they face wear-out effects which will result in lowered write endurance of the device. Similar to Flash based SSDs, SCM cells have a finite number of writes before failing. Accordingly, it is difficult to scale-out storage systems based upon these technologies. Even single systems which use SCM main memory would fail after brief use. Accordingly, techniques to avoid catastrophic failures need to be developed to make SCMs feasible as DRAM replacement. Techniques such as RAID, which replicate data over multiple devices to improve failure tolerance by increasing redundancy, have to be adopted to operate at the timescales of SCM access times. However, even such a system would require different techniques as RAID relies on a controller that poses an undesirable single point of failure.

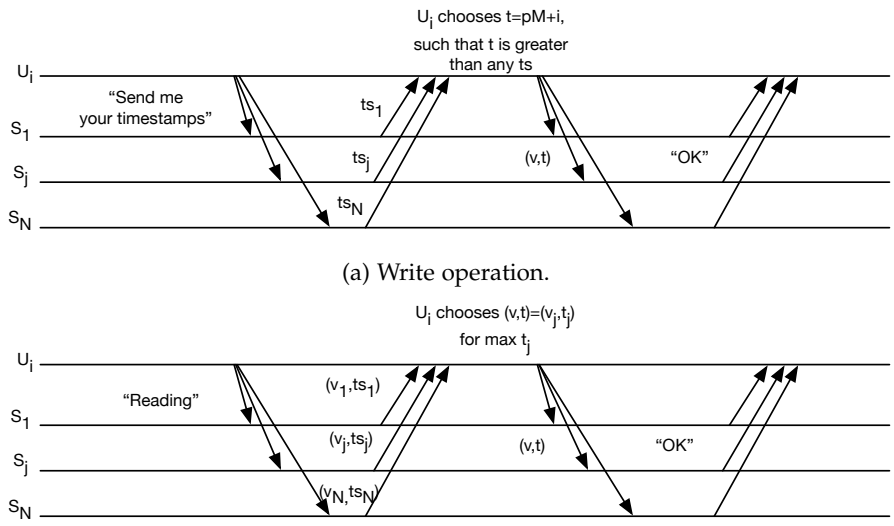
Traditional memory hierarchies were able to ignore errors that occurred. The caches and DRAM was treated as being flawless and any error would lead to a system failure, going as far as being the single most common cause for crashed systems [108]. In general, this approach is not sufficient for larger memories [30]. Techniques to reduce memory errors such as ECC exist, but come at a high cost and result in lowered performance.

A very expensive approach is employed in contemporary supercomputers. Due to the large number of components that can fail at any time, the systems employ a *checkpointing* mechanism. At different times, the content of the DRAM is stored on non-volatile memory. After a crash, the system can be restored to the checkpointed state and continues working. These systems are not only very expensive but also require complex management to reduce the incurred overhead [85].

This chapter presents a different approach to fault-tolerance in non-volatile main memory based on SCM. Memory in this system is treated as *distributed* storage and data consistency is ensured through a *consensus protocol*.

Traditionally known as a severe bottleneck to access performance, the new generation of programmable switches [16, 65, 129] and FPGAs enable high-performance, low-latency consensus in the network [27, 29, 60, 63, 79, 98]. The consensus logic is executed directly in the network leading to large latency reductions and increased throughput.

The consensus algorithm employed here is a generalized version of a protocol by Attiya, Bar-Noy, and Dolev [8], hereafter called the ABD



(a) Write operation.

(b) Read operation.

Figure 9.1: The ABD protocol as introduced in [30]. Reads and writes require two phases to determine which state is the current one and perform the requested operation.

protocol. The protocol ensures linearized fault tolerant read/write access to the memory. At the same time, it is less complex than other consensus protocols such as Paxos [27], which is important for an efficient implementation in the utilized hardware switch. Furthermore, the state required to be stored on the switch is low during protocol execution, accommodating for the low availability of switch resources to keep state such as SRAM.

This chapter is organized as follows. Firstly, the ABD protocol as well as the PCM employed is introduced (Section 9.1). Secondly, the mapping of the algorithm onto the hardware is described (Section 9.2). Lastly, the system, based on ASICs and FPGAs, is evaluated in Section 9.4.

9.1 BACKGROUND

This paper deals with an SCM type called Phase-Change Memory. The background and characteristics are explored hereafter. Additionally, the ABD protocol, as it is used in this work, is introduced.

PHASE-CHANGE MEMORY (PCM) To this date many different SCMs have been proposed. The technology that is commercially most successful is Phase-Change Memory (PCM) [125]. The technology has been employed initially in low powered mobile devices [109], and is nowadays also used in enterprise storage [35]. Alloys of Germanium, Antimony, and Telluride (GST) are the basis of the memory elements. Quickly heating and then cooling the material results in "an amor-

phous solid state with high resistivity and good optical transparency" [30]. Heating the material just below the critical melting temperature results in “an opaque solid state of low resistance” [30].

These GST materials have already been employed since the late 1960s [93] in optical storage media such as BlueRay. Using GST for solid-state memories, on the other hand, is a new development, requiring suitable *selector* devices [19, 96, 110] to form larger arrays of memory cells with better die utilization resulting in lower costs of the technology. PCM can be bought in different products such as Optane® and 3D XPoint® from Intel and Micron [67].

The response times of PCM are very fast and typically in the order of 100 ns. In laboratory conditions, access times below one ns have been achieved [80], which makes PCM comparable to DRAM in access latencies. Flash based SSDs have access times many orders of magnitude larger, typically in the range of 50 µs to 70 µs, without protocol overheads and closer to 100 µs with the required error correction and protocol overhead [30]. Another advantage compared to flash is PCMs *byte-addressability* for reads and writes, which, for instance allows for many optimizations in database systems. Flash on the other hand requires erase block management and garbage collection to deal with the fact that Flash cells have to erased before writing new data to them [118]. Compared to DRAM, PCM offers non-volatile memory with low latencies. Non-volatile memories with a similarly low latency, such as battery-packed DRAM, have to be constantly powered to retain state, increasing their cost and maintenance complexity. PCM retains data for many years and has a high write endurance in the millions of cycles range. Furthermore, PCM is cheaper to manufacture than DRAM because of its simpler memory cell structure and denser packing.

ABD PROTOCOL The protocol used subsequently is described in [8] by Attiya, Bar-Noy, and Dolev. As the author did not name their algorithm, the shorthand ABD, which is derived from the first letter of either name, is used. The protocol implements an atomic register in an asynchronous message-passing system. An atomic register is defined as a register that can be accessed through a read and write function. The functions guarantee that every read "returns either the last value written or a value that is written concurrently with this read" [8]. Similarly, a read operation R_2 that is started after another read R_1, can not return an older value that R_1. The protocol is optimized for read and write requests which makes it a good candidate for a storage class memory utilizing those functions. Competing protocols such as Paxos [74] and Chain Replication [100] allow arbitrary operations, such as increments, but allow so at the cost of additional communication steps.

User processes perform read and write operations on a shared memory through message channels. Accordingly, the protocol emulates shared memory with message passing [30]. In [8] only a single writer

is assumed, but the protocol can be generalized to support multiple writers. The modified protocol works in the following way.

As stated in [30]: "The ABD protocol assumes there are M user processes, and N server processes. Every user process can send a message to every server process, and vice-versa. Each user process $U_i \in \{U_1, \ldots, U_M\}$ chooses a unique timestamp of the form $t = pM + i$, where p is a positive integer. For example, if $M = 32$, U_1 chooses timestamps from the set $\{1, 33, 65, \ldots\}$." The naming convention helps identifying which user process issued any given request. Figure 9.1 shows the two phase process for read and write operations in ABD.

A write of value v is initiated by the user process U_i requesting the current timestamps of all server processes. The server processes $S_j \in \{S_1, \ldots, S_N\}$ respond on that message with their current timestamp ts_j. After receiving the majority of the timestamps from S_j, U_i choses a new timestamp t in the form $t = pM + i$, so that t is larger than any ts_j received and the original t. After determining a suitable candidate t, U_i sends the pair (v, t) to all server processes. The server processes have to compare the retrieved value of t to their local timestamp ts_k. The value v is only written if t is greater than ts_k. If this is the case, the local timestamp and value are overwritten with t and v. Lastly, an acknowledgment is returned to U_i by the server processes to signal write completion.

Performing a read works very similar to a write. The first phase determines the current state of the memory cell and the second phase updates any servers that are not up to date. Accordingly, initially U_i sends a read message to all server processes. The server processes $S_j \in \{S_1 \ldots S_N\}$ respond with the current value and timestamp (v_j, ts_j). The user process U_i then determines which tuple $(v, t) = (v_j, ts_j)$ is the most recent one after receiving a response from the majority of servers. The most recent tuple is the tuple where ts_j has the highest value. For the second phase, U_i sends the tuple (v, t) that has been determined to be the most recent back to all server processes. The servers process the tuple as they would with a write and update their local value and timestamp if the received timestamp is larger than the currently stored one. Lastly, the servers acknowledge the operation back to U_i.

The protocol itself does not have a notion of addresses and deals only with a single memory cell. For larger memory cells a mechanism has to be added to perform the algorithm separately for any memory cell. The next section describes the changes necessary to implement the protocol on the utilized hardware.

9.2 DESIGN

The high level view of the system is presented in Figure 9.2. The central element is the programmable switch, which is either an FPGA

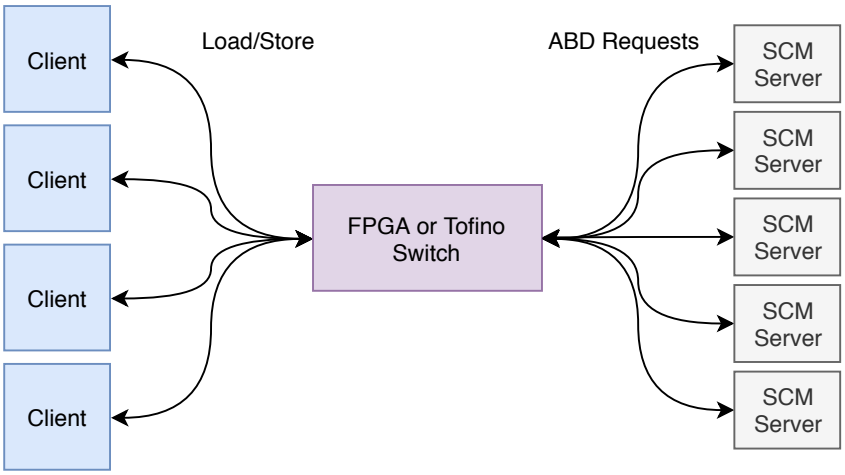


Figure 9.2: Memory access requests by the clients are translated by the programmable switch into ABD requests. The switch is responsible to perform the necessary operations for the protocol.

or a Barefoot Tofino based switch. Both are running the modified ABD protocol and ensure that the memories stay consistent. The switch acts as the user as described in Section 9.1. The clients on the left of the illustration issue read and write requests to the programmable switch. The clients are implemented inside an FPGA using a custom memory controller. Lastly, the SCM instances serve as the servers as introduced in Section 9.1 and are also implemented inside multiple FPGAs.

Two implementations of the ABD functionality were done. Both implementations come with stringent performance demands on memory accesses. The first implementation uses P4 to put the protocol inside a Tofino ASIC. Secondly, an FPGA based implementation is compared to see what a custom architecture could achieve compared to a general purpose switching ASIC.

The hardware characteristics of both platforms, but especially of the ASIC, introduce certain limitations:

- The Tofino switch has limited memory available in each stage for stateful operations [111].
- The switch is limited to one arithmetic instruction per field for each stage of the pipeline.
- There is a limited size of the state that can be passed between stages.
- The maximum length of the pipeline is limited as there is a fixed number of match units.

None of these limitations apply fully to the FPGA implementation. The FPGA can leverage more on-chip memory and has access to off-chip

DRAM. Additionally, the architecture can be completely customized which removes the other three limitations from the FPGA.

Accordingly, the implementation has to consider the physical implementations of the ASIC to be efficient. The following assumptions were made to take care of these issues:

- The goal is to integrate the system into off-the-shelf hardware. Accordingly, integration should be as painfree as possible. The requester, which will later later become, for example, the CPU cache controller, should not need to know about the consensus protocol. Accordingly, only simple read and write requests are sent but no ABD logic is implemented inside the client.
- Caches work on cache lines of a certain size. The protocol addresses this by utilizing 64 bytes, a typical size for cache lines, as its value.
- This is an initial prototype which assumes that the switch never fails. Accordingly, complete fault-tolerance can not be addressed right now, as the device might fail. Considering that the mean time to failure for memories is orders of magnitudes shorter than the mean time to failure for the switches, the trade-off for a simpler protocol is taken.
- The protocol does only support direct connection of clients and servers to the switches. More complex topologies are not supported. This limits the applicability in practice but greatly simplifies the protocol for the proof-of-concept.
- The scenario consists of ~1000 CPUs issuing 10 concurrent requests each. The total load on the switch is approximately 10 000 requests concurrently.

The following sections describe the individual parts of the system starting with the memory controller which acts as the client.

CLIENT MEMORY CONTROLLER The memory controller is implemented inside an FPGA using the Bluespec network packet parser presented in Section 7.1. The memory controller is kept as simple as possible and aims to reduce latency as much as possible. The read and write requests are configured by the host via control registers. The request is then translated into an ABD network packet and transmitted via the network interface. The interfacing is realized using TaPaSCo with the SFP+ plugin. Incoming response packets are parsed and the pending AXI request is answered. Accordingly, a single AXI request issued by the host corresponds to a single request over the network to the consensus system.

On the host, the ABD reads and writes should work as transparently as possible and without modifying user applications. This is realized

through a special device driver that performs the translations by handling page faults. This is realized by intercepting the `malloc` and `free` system calls to call custom versions instead of the standard system calls. These custom implementations `mmap` the char device provided by the driver and issues requests through that region. For an allocate operation, the driver issues a remote allocate of the requested size on the server. The address is returned to the client driver to transparently perform the translations.

The user application can then access the shared memory transparently through the shared region. To realize this, the driver maintains a local buffer with configurable size to serve page faults. This buffer is typically the same size as a page (here 4 kB). A page fault is handled by the driver by fetching the page via the FPGA from the remote memory. This page cache works with a write back approach, which means that changes to the page will be written back only when a new page is requested. The changed content will be written back to the remote server and the new page is requested afterwards.

SERVER MEMORY CONTROLLER The server side is implemented based on the same Bluespec network packet parser. Incoming packets are parsed and if they are requests belonging to the protocol they will be answered accordingly.

A packet can be either of the following: (1) Memory Read, (2) Memory Write, (3) Timestamp Request or (4) Writeback Request. For the implementation requests (1) is the same as (3) and (2) is the same as (4) with minimal changes to the timestamp field of the response, simplifying the parser. The parser performs the following operations:

1. Drop packets that are neither ABD packet nor for myself.
2. Retrieve requested data from memory. Data contains timestamps and value. Necessary for both reads and writes.
3. Process request
   a) *Read and Timestamp:* Return the value and timestamp of the requested address.
   b) *Write and Writeback:* Update memory location depending on own timestamp. Return latest known version.

Accordingly, the memory can be implemented very efficiently and in small hardware. Memory delays are accounted for by using buffers that contain the necessary information for later stages.

GENERAL SWITCH LOGIC Additional changes to the protocol in addition to those introduced in Section 9.1 are necessary to fulfill the assumptions made previously. The protocol is generalized to serve *multiple* registers instead of just a single one as is the case for the original version. Each register corresponds to a single cache line and

comes with a timestamp. The user as introduced in Section 9.1 is the switch. The clients are not performing any ABD related processing as the memory accesses should be transparent.

The switch has to keep some state that depends on parameters of the system. The number of cache lines stored are determined by the size of the memory and the cache line size:

$$\#\text{ of cache lines} = \frac{\text{size of address space}}{\text{size of cache line}} \tag{9.1}$$

Accordingly, with 4 GB of memory and 64 B per cache line, the system stores 67 108 864 cache lines.

P4 IMPLEMENTATION The P4 implementation has to consider limitations of the hardware. State is kept in "registers", which are arrays of cells. Each cell is sized according to the ALUs of the underlying hardware. Accordingly, each cell is usually smaller than a cache line and a single cache line is split across multiple register cells. The number of cells can be calculated as

$$\text{cells per cache line} = \frac{\text{size of cache line}}{\text{size of cell}} \tag{9.2}$$

The limited available memory is a problem if each cache line comes with its own timestamp, and the address space is large. Instead of giving each cache line its own timestamp, a block of *multiple* cache line receives a time stamp. Overall, the number of cache lines and the number of timestamps must be less than the total memory available:

$$\begin{aligned} &((\#\text{ of cache lines} \times \text{entries per cache line}) \\ &\qquad + \#\text{ of timestamps}) \times (\text{size of cell}) \\ &\leqslant (\text{memory per stage}) \times (\#\text{ of stages}) \end{aligned} \tag{9.3}$$

For instance, a cache line is 64 bytes and the switch can allocate 32K registers for 64-byte cache lines and 32-bit timestamps. As a result, a block of 2K cache lines has to share a timestamp to fit 4 GB of data.

Moreover, the switch code uses an additional 4 registers, each with (# of timestamps) cells of size 8-bits to keep track of the information returned by the servers. The state is used to determine if a majority has been reached in each of the algorithm steps.

The P4 implementation does packet forwarding at Layer 2.

Addressing of the memory is done through Ethernet multicast. One multicast group is assigned to each set of replicas. Sending messages to the replicas is done by setting the corresponding MAC address of the multicast group identifier as the destination.

FPGA IMPLEMENTATION Contrary to the many limitations that come with the off-the-shelf hardware, the FPGA does not have any of

these limitations. No fixed parser stages means no limited memory for any given stage. External memory increases the available storage space by many orders of magnitude. Accordingly, the FPGA implementation would not need the limitation on multiple blocks of cache lines per timestamp. However, to make a fair comparison, the FPGA implementation also uses blocks of cache lines. This restrictions limits the number of parallel requests, as there can only be one request per timestamp at any time.

The implementation itself uses the same parsing library as the memory and client does. Incoming requests are duplicated to the outbound ports and translated into the corresponding ABD packets for phase one. Received answers are accumulated until a majority has been received and then retransmitted for phase two. The answer to the client is already transmitted after completion of phase one. The current implementation keeps track of the active requests in BRAM and allows 65 535 concurrent requests.

FAILURE ASSUMPTIONS The ABD protocol assumes only that the a majority of participants is available. Accordingly, the protocol will continue to work as long as the required number is reachable. This assumption can be compromised if the switch itself fails. Switch failure can be dealt with by the introduction of redundant components to replace the faulty switch. Furthermore, the protocol has to be extended to include the possibility of a backup switch. The prototype presented here does assume that the switch will never fail. In comparison, Paxos relies on the election of a non-faulty leader if a failure state has been detected [74].

Another problem experienced in networks is packet loss. For now, a time out is used to determine if a packet has been lost.

9.3 IMPLEMENTATION

The P4 based switch for the client side is implemented in 858 lines of $P4_{14}$. The code runs on a Barefoot Network Tofino ASIC and has been compiled with Barefoot Capilano [16]. The particular model is a 32-port top-of-rack switch. All FPGA based components run on Xilinx NetFPGA SUME, which contain four SFP+ ports and a Virtex 7 based FPGA and 8 GB of DDR3 memory.

The driver that implements the page fault handler on the client side is written in 1157 lines of C code. As described in Section 9.2, the driver handles page faults by requesting the pages from the server and answers returns the requested data back to the user space application.

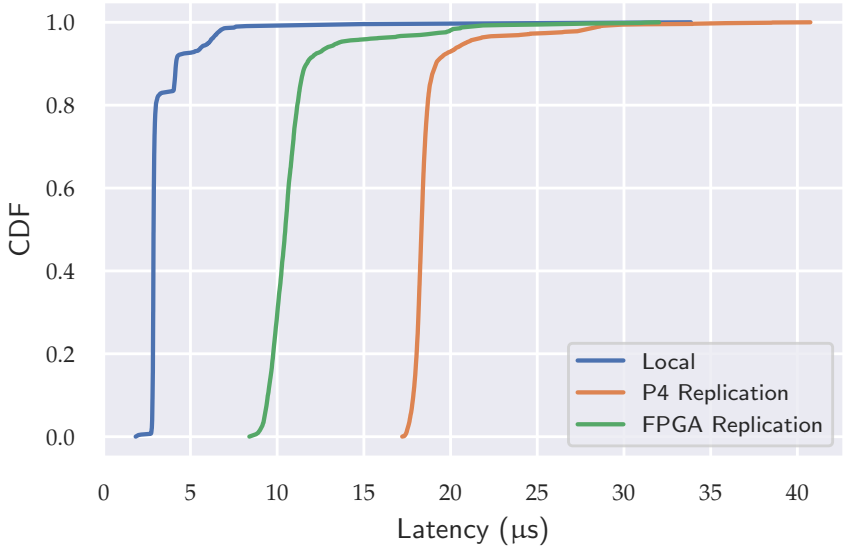


Figure 9.3: CDF of the latency measured for the different methods when reading a cache line. Local reads the value from local memory, P4 Replication reads the value from the remote memories via the P4 switch and FPGA Replication reads the values via the FPGA switch.

9.4 EVALUATION

The evaluation presented at this point focuses on measuring the overhead incurred by page fault handling by the remote memories.

The switch running the ABD protocol is configured to run at 10G per port. The memory endpoints are implemented on three Xilinx NetFPGA SUME FPGAs, another one is used as a switch if the Tofino is not in use. The host running the client part is a dual-socket Intel Xeon E5-2603 with 12 cores running at 1.6 GHz, 16 GB of 1600 MHz DDR4 and a Intel 82599 10 Gbit/s NIC. The individual parts are connected using 10G SFP+ copper cables. The servers are running Ubuntu 16.04 with Linux kernel version 4.10.0.

The preliminary experiments shown here do not use a true memory controller in hardware. Instead, the behavior is emulated using an application that calls `mmap` to map a file into memory, and then issues write requests to addresses at different pages. The time it takes to execute a single request is measured. The experiment has been repeated 100 000 times for each of the configurations. The first one does not request a new page over the network but uses local memory instead. The second one uses the FPGA as a switch, and the last one employs the P4 based implementation to interface with the remote memories.

The median latency for the local approach is 3 µs as shown in Figure 9.3. Fetching over the P4 switch requires 18 µs. The FPGA is faster and requires only 10 µs. One caveat is that the P4 switch implementa-

tion has to perform full Layer 2 parsing, a custom protocol can further reduce the latency for the off-the-shelf switch. Accordingly, these initial results are very promising. The experiments achieve lower latencies compared to traditional replicated storage systems. The complete system looks very suitable for use with scalable main memory.

9.5 CONCLUSION

In conclusion, Storage Class Memory, offers the potential to disrupt the traditional memory hierarchy [30]. In-network consensus solves the problem of the limited write endurance of these new memories. Even the preliminary tests without hardware accelerated clients already shows very promising results. Over the network, access times are very stable at around 10 µs, not even an order of magnitude slower than conventional memory, and require just 55 % of the time than on a state-of-the-art programmable switch.

10 SPIKING NEURAL NETWORKS

The work presented in this chapter has been published in:

1. Amir Zjajo, Jaco Hofmann, Gerrit Jan Christiaanse, Martijn Van Eijk, Georgios Smaragdos, Christos Strydis, Alexander de Graaf, Carlo Galuzzi, and Rene van Leuken. "A real-time reconfigurable multichip architecture for large-scale biophysically accurate neuron simulation." In: *IEEE transactions on biomedical circuits and systems* 12.2 (2018), pp. 326–337

2. Jaco Hofmann, A. Zjajo, and R. van Leuken. "Multi-chip dataflow architecture for massive scale biophysically accurate neuron simulation." In: *38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. 2016

3. Jaco Hofmann. "Real-Time Multi-Chip Neural Network for Cognitive Systems." In: ed. by A. Zjajo and R. van Leuken. Real-Time Multi-Chip Neural Network for Cognitive Systems. River Publishers, 2019. Chap. MultiChip Dataflow Architecture for Massive Scale Biophysically Accurate Neuron Simulation

Spiking Neural Networks (SNNs) are a method for simulating neuron cells and the interaction between them using mathematical models of their biophysical makeup [41, 61]. As SNNs model the chemical behavior of neuron cells, they simulate more than just the firing rate of neurons, as Artifical Neural Network (ANN) do. SNNs also include the amplitude of spikes, spike train patterns (See Figure 10.1 for an example), and the transfer rate [134].

A spike train is a representation of the neural activity. It is a sequence of action potentials, which means the times when the neuron fires [41].

Thanks to their high accuracy, SNNs can be used for brain operation research, without relying on in-vivo experiments. However, the accuracy results in a high computational cost. The Hodgkin-Huxley (HH) model [50] is a very accurate representation of the internal state of neuron cells and the connectivity between them [134]. The model requires a large number of double precision floating point (FP) operations. To put this in perspective: Calculating a single neuron cell with no connection to another cell requires 859 FP operations [134]. Adding one connection to another neuron requires an additional twelve FP operations.

The simulation needs to be done very quickly and very accurately to meet research demands. This requires that double precision floating point is used and that the simulation is fast enough to meet brain real-time. The brain real-time in this context is defined as 50 µs per simulation step [134].

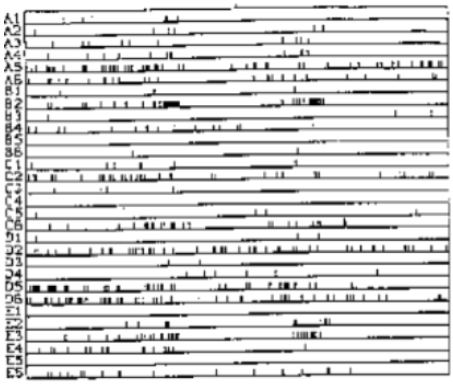


Figure 10.1: Example of a spike train of thirty neurons from a monkey cortex. Time is plotted on the vertical axes, while the vertical axis represents the spikes. Taken from [70].

As most of the calculations are independent from one another, the system benefits from a high degree of parallelism. FPGAs have been shown as promising candidates to meet the accuracy and speed requirements. In [113] a Virtex 7 based system is proposed that meets brain real-time for up to 96 neurons. Their approach could be scaled up to more cells from a purely computational aspect. However, the approach is limited by the exponentially increasing communication cost, which is a result of the bus that is used for communication.

This chapter introduces a system to simulate neurons of the inferior olive nucleus (ION), which is a very well charted part of the brain [134], on FPGAs. It focuses less on the biological aspects, as they are introduced in [134], but more on the system performance and the intricacies of using a high level language, in this case SystemC, for system modeling.

10.1 SYSTEM DESIGN

The system as presented in [55] is built from simulation engines, called Physical Cells (PhC). The PhC are responsible for calculating the simulation steps for a single neuron. These calculations include the behavior of the neuron itself, as well as all incoming communication from connected neurons. As the calculation is short compared to the brain real-time, it can be done multiple times per PhC in the required 50 µs. For instance, the calculation of a single neuron at 100 MHz requires only a about 528 cycles, or 5.28 µs, per simulation step. Hence, each PhC can serially calculate *multiple* neurons in each step. The number of cells each PhC calculates per step is called the Time Share Factor (TSF).

The neurons have to communicate in some way. The optimal case would be that all neurons can communicate in a single cycle with any other neuron. This can not be done for routing and resource reasons.

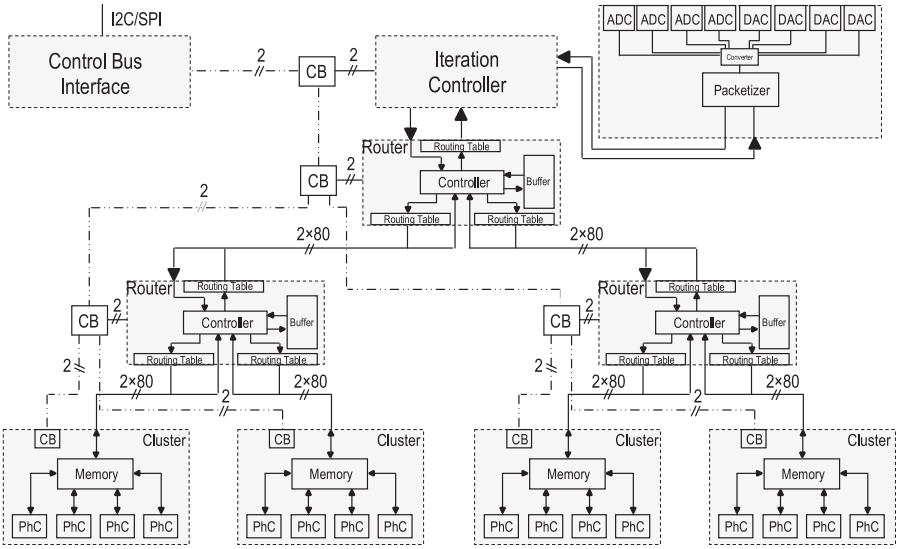


Figure 10.2: System overview of the proposed SNN simulation architecture. Clusters contain multiple PhC around a shared memory. Each PhC serially calculates the data for multiple neurons. Clusters are connected via a tree NoC to one another. Clusters that are close together can communicate faster. This models the connection schemes of neurons found in the brain. Taken from [134].

Nonetheless, the system tries to let as many neurons as possible communicate in a single cycle. This is realized by attaching multiple PhC to a shared memory to form a cluster. The PhCs have time shared access to the memory. Hence, the cells in a cluster can communicate directly with one another. The optimal number of PhC and the TSF in one cluster have to be determined based on the FPGA technology and memory used. A larger number of PhC in each cluster simplifies communication, but increases routing delays on the FPGA. In addition, each PhC has to wait longer for memory access.

The clusters are connected through a custom Network on Chip (NoC) that makes use of the neuron connection characteristics. Neurons are not equally likely to be connected to one another. Instead, neurons that are spatially closer together are more likely to be connected. Furthermore, neurons are not connected to every other neuron, but rather with only about 10 % of their neighboring neurons [18]. Hence, the clusters are connected using a tree architecture. Neurons that are located close together, but not in the same cluster, can communicate over fewer hops than neurons that are farther apart. In addition, the tree does not have to be a fat tree, which grows thicker further up the tree. This is the case, because communication over the upper layers of the tree are less frequent.

The complete system is shown in Figure 10.2.

The routers of the NoC are responsible to forward cell information to other connected cells. Routing is based on precalculated routing tables. Each output of each router contains one routing table. A packet

received by the router is checked against the table and is forwarded if the sender is listed in the table. Hence, the system uses a sender based routing protocol. This has the advantage that no information but the sender and the data has to be transmitted.

The execution is controlled at the top of the tree by an iteration controller. Each PhC signals completion to its cluster, and each cluster then signals completion up the tree to the controller. The iteration controller is responsible for ensuring that the system keeps pace.

The top of the tree also contains the input and output stages. Analog-to-digital converters are employed here to sample signals from the outside world and feed the signals as inputs into the tree network. The network is responsible for moving the inputs to the correct neurons based on the precalculated routing tables. Data output is handled in the opposite direction, when cells forward their outputs to the top of the tree.

Another design requirement is that the parameters of the cells, and of their connectivity, can be changed during simulation. Communicating with every component in the system is possible through a custom bus system that follows the structure of the tree to reach every component. Firstly, the address is transmitted. The address is used to open up a channel to the destination component. Secondly, the data is transmitted which is interpreted by the destination to perform the desired operation. The bus does not provide high throughput, but throughput is no primary concern as the system is halted during reconfiguration.

10.2 SIMULATION

The proposed system is written in SystemC and can be simulated cycle accurate. The simulation is used to make design decisions. In this case, the number of PhC per cluster and the fan-out of the tree are determined based on simulation.

ROUTER FAN-OUT The first parameter that has to be determined is the size of the routers. Larger routers bring more cells closer together but experience heavier traffic as a result. Small routers are more efficient but can only connect few clusters. Hence, the simulation is used to determined the optimal router sizes. The number of neurons in the simulation is kept at 512 for all simulations. Furthermore, all neurons are connected to one-another which is the worst case scenario for the network.

The first simulation presented in Figure 10.3 shows that smaller fan-outs perform better than larger fan-outs. The plot shows the number of cycles needed to calculate a single simulation step. This time includes the time needed for communication between the cells. The optimal fan-

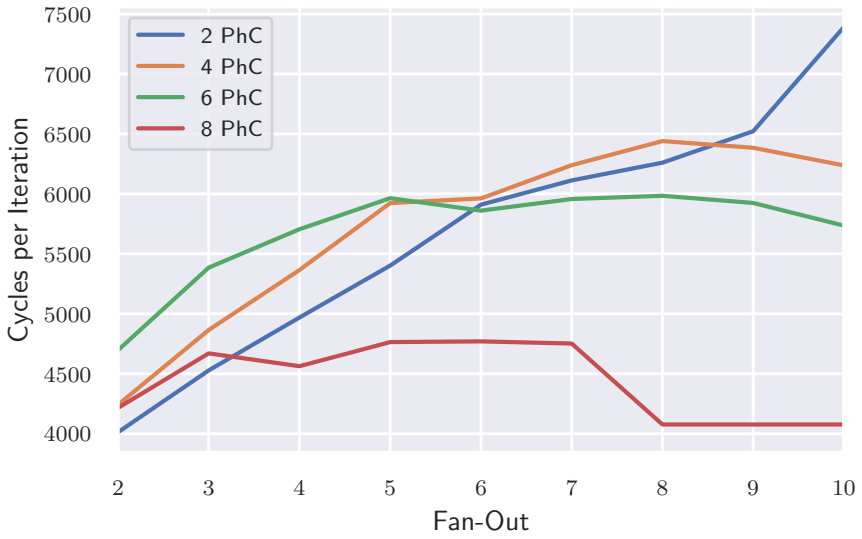


Figure 10.3: Cycles needed for one simulation step at different router fan-outs and for different number of PhCs per cluster. Smaller fan-outs tend to perform better as the routers are less crowded.

out for most cluster sizes is two. The only exception can be observed at eight PhCs per cluster, where the optimal fan-out is at eight to ten.

PHC PER CLUSTER As two seems to be a reasonable number for the fan-out, the next question is how big the clusters should be. The simulation continues to use the all-to-all connection scheme. The simulation should show if the faster communication between cells in larger clusters can hide the performance lost by the slower memory access.

The simulation results in Figure 10.4 show that smaller clusters are favorable. The optimal result is reached when using just two PhC per cluster. The difference at 2048 neurons between the two and 18 PhC per cluster configuration is 30 %. The two, four and eight PhC configurations perform almost identically.

SYSTEM PERFORMANCE Based on these results, the system can be compared to the baseline as presented in [113]. The baseline itself uses a shared bus to connect the PhC with one-another. The communication cost in this system growth exponentially with the number of neurons.

For this comparison, the neurons are connected only to their direct neighbors, instead of the all-to-all connection scheme used for the other scenarios.

The results in Figure 10.5 show that the scaling of the proposed system is linear instead of exponential. Hence, the new system is able to meet the brain real-time requirements with a much greater number of simulated cells.

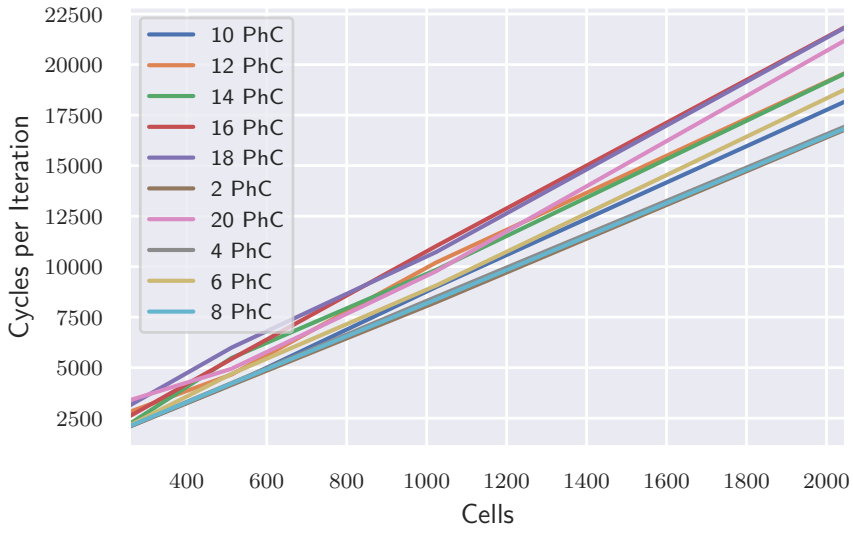


Figure 10.4: Cycles needed for one simulation step at different cluster sizes. The router fan-out is kept at two as determined in Figure 10.3.

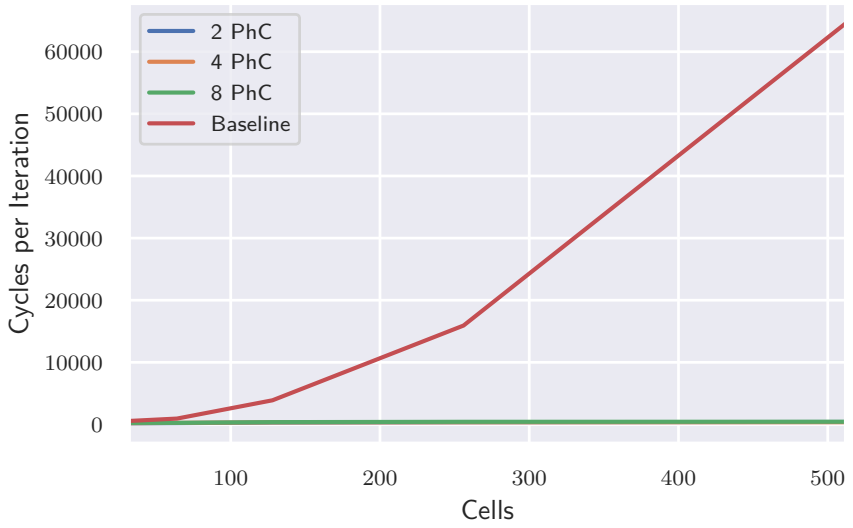


Figure 10.5: Cycles needed to complete one iteration of the full HH model for the given number of cells. The baseline is taken from [113]. The improved interconnect on the proposed system results in linear scaling with the number of notes, compared to exponential scaling of the original baseline.

Whereas the baseline was peaking at 96 simulated neurons, the proposed system can simulate orders of magnitude more cells. When each neuron is connected to only its immediate neighbors, up to 19200 cells can be simulated while meeting brain real-time. About 3000 cells are possible in a realistic scenario of about 10 % connectivity.

10.3 MOVING TO HARDWARE

The simulation shows promising results and the next step is to move the design onto an FPGA to confirm the findings. The complete system thus far is written in SystemC 2.2. Accordingly, it should be no problem to use HLS synthesis tools using the same model for simulation and synthesis, as it is advertised by the tool providers.

Unfortunately, the tools were not there, yet. While the code was compatible to the SystemC specifications, it was not compatible to the small subset that VivadoHLS 2014.1 used. Accordingly, the complete code had to be rewritten to the the dialect that was understood by the tools. As mentioned for C and C++ in Chapter 3, this results in very different code for SystemC as well. As many features of SystemC could not be used the resulting code resembled an HDL more than a HLS language.

Overall, starting out with a HCL would have resulted in shorter development time. The experience from other projects shows that initial development of the simulation can be faster using SystemC, but the development of the hardware prototype is faster using HCL directly. If the tools for SystemC catch up with the language itself, this balance can tip over in favor of SystemC.

Nonetheless, designing using SystemC was quite pleasant. Changes to the design were relatively pain free compared to using a HDLs. The inclusion of C++ features for simulation and designing simplified things further. One example of this was the use of interfaces between modules that contained a variable number of ports based on template arguments. This feature is used to dynamically change the fan-out of routers.

10.4 EVALUATION

The evaluation is performed on a Xilinx Virtex 7 XC7VX550 FPGA based on the adopted SystemC design. Apart from the HH model, two simpler models are evaluated for comparison. The Izhikevich [61] and Integrate & Fire [112] models require less computations which means that more cells can be time shared per PhC.

The synthesis results show that all three scenarios are limited by the number of LUTs. The system containing the fewest neurons is unsurprisingly the HH model, because the model needs to most floating point operations. The system contains 16 clusters housing two

Table 10.1: Configuration and hardware utilization for the system and three different neuron models as reported in [134]. All systems run at 100 MHz.

Model	Cluster	PhC	TSF	BRAM%	DSP%	FF%	LUT%	Neurons
Hodgkin-Huxley	18	2	33	23.6	35	27.5	90	1188
Izhikevich	5	8	70	38	22	25	89	2800
Integrate&Fire	5	8	75	23	20	16	54	3000

PhC which time share 33 neurons each, resulting in 1188 total cells. For the faster models, the number of time shared cells goes up as the number of cycles required per cell decreases. Using the Izhikevich system, the design consists of 2800 cells with a configuration of five clusters with eight PhC each. Lastly, the Integrate&Fire model can be simulated with 3000 cells.

In total, the proposed system increases the number of high accuracy neurons by 12.375×.

10.5 CONCLUSION

This chapter presents a system that uses FPGA to simulate accurate models of neurons while keeping up with tight real-time constraints. Such a system can be used to study the behavior of parts of the brain, replacing in-vivo experiments, enabling much deeper insight into brain operation. Furthermore, the system allows tight control of the simulation parameters and can even be used to change concentrations of different chemical in the neurons. In addition, the connectivity between the neurons can be changed during run-time which can be used to research learning.

The simulation based on a SystemC model shows that between 3000 and 19200 cells can be simulated while keeping up with the real-time requirements. Previous attempts are limited to only 96 simulated cells, due to exponential growth in communication cost [113]. This performance is achieved through a custom NoC that is optimized for the communication patterns found inside the brain.

The preliminary hardware results show that the system can simulate 1188 neurons on one medium sized FPGA. Using faster, and less accurate, models the system can contain even more neurons. The largest system implemented on the one FPGA contains 3000 neurons using the Integrate&Fire model.

11 CONCLUSION AND LESSONS LEARNED

This thesis introduced FPGAs as a target for accelerator development. Whereas the improvement potential using FPGAs is high, there are still many open research questions that limit the current main stream appeal of FPGAs. They are difficult to program as they require a completely different mindset when targeted directly using HDLs or HCLs. Alternative approaches of using software programming languages such as C, C++ or OpenCL often fail due to tools that offer only limited language support and are often unstable [114]. The tool chain problem goes all the way down to the synthesis and place&route tools that are offered by the FPGA vendors as closed source applications. A common occurrence till this day is that the toolflow crashes during execution without any explanation. Debugging those issues becomes increasingly difficult and requires a lot of trial and error. As the tool chains are not necessarily deterministic and use randomization during their runs, the problem might go away by simply rerunning the same run, but there is no guarantee for that.

However, the major vendors noticed the problem and nowadays open up their tool chains and provide new DSL based tools. Projects such as Xilinx Vitis [127], that was announced on the first of October 2019, try to bring FPGAs directly into the relevant frameworks. Supported frameworks include TensorFlow [2] for deep learning applications and FFmpeg [13] for video decoding. The framework shows how a wider audience can benefit from FPGA, as they provide a flexible platform for hardware acceleration of a wide number of tasks.

The mainstream appeal right now is reduced by the limited availability. Most FPGAs are very expensive and therefore currently not bought by consumers. However, this might change in the not to distant future. Apple launches their new Mac Pro with an accelerator card called the Afterburner. The card is based on an undisclosed FPGA and promisses to accelerate video editing task for 8k footage. Whereas, the card will be single purpose at launch, Apple might open up development for other accelerators later on. The Intel Hardware Accelerator Research Program (HARP) [5] puts FPGAs inside server CPUs with full cache coherent access to data. Hence, FPGA could become available in most CPUs at some point.

Comparing the situation to early GPUs: They did not become popular because they could be used for complex computations. Instead, they were the driving force behind the computer games market. Accordingly, most people already had a GPU when GPUs became more flexible and were increasingly used for number crunching. Interested

parties already had a GPU at home and could simply experience GPGPU, which was the foundation of the GPU boom.

In addition, access to FPGAs can now be rented in the cloud which makes them more accessible than half a decade ago. Servers with attached FPGAs can be rented for a small fee from many big server providers, such as Amazon's F1. These instances also provide the required tools and infrastructure to get started with FPGA.

Nonetheless, FPGA are not that useful if they can not be used for common applications. Part II shows that FPGA can provide great improvements upon off-the-shelf hardware for different problem domains.

In conclusion, FPGA themselves and architectures that perform well on them are a very interesting research subject. There are many open questions, but the landscape is slowly changing from a very closed source and small community to an open community with mainstream appeal. FPGA have the potential to create efficient data centers that are not fixed function, but can be adopted to novel applications through dedicated accelerators.

11.1 LESSONS LEARNED

To end this thesis, the following section contains some more general lessons I have learned during my time developing for FPGA.

The first and most important lesson is that nothing is easy in hardware. FPGAs are hard, and the developer should take every advantage available to tame them. Testing a design should be done right away, even when it seems like an easy design. Too often the claim "This can not be so difficult" resulted in deployment without proper testing, which in turn led to tedious bug hunting because of the hubris of the developer.

Languages such as Bluespec assist the developer in writing complex testbenches. Projects such as QuickCheck [23] can provide automated testing of code in software languages. A reimplementation for Bluespec is available as BlueCheck [88]. This method of testing is especially useful in hardware debugging and finds many issues that were not found by hand-crafted test benches. BlueCheck will even try to find the shortest way of replicating the problem.

For instance, often times a functional model of a hardware component exists. This model is known to be working correctly, but is not very efficient. Another implementation that is very efficient but untested can be compared to the known good model using BlueCheck. In this case, BlueCheck checks the equivalence of the two implementations as shown in Listing 11.1. If BlueCheck finds an error it will print the execution trace of the functions and tries to find a shorter trace to replicate the error if desired.

```
/* Equivalences */
equiv("pop", s1.pop, s2.pop);
equiv("push", s1.push, s2.push);
equiv("isEmpty", s1.isEmpty, s2.isEmpty);
equiv("top", s1.top, s2.top);
```

Listing 11.1: BlueCheck can check the equivalence of two stack implementations by using random testing. The library will try to find counter examples where the two implementations behave differently. BlueCheck then tries to find the shortest path to replicating the issue.

When using languages such as Bluespec, the developer should make sure to develop libraries to promote code reuse. Reusing code in high-level languages such as Bluespec is much easier compared to Verilog. The interfaces are clearly defined and reusing the modules of other designers is as easy as including the correct package. Furthermore, standardized interfaces allow implementation changes without the need to change anything else about the design. This practice is commonly employed in Bluespec when it comes to FIFOs. Initially, the default FIFO type of Bluespec is used which supports only two entries. Later on, the FIFOs can be replaced by specialized implementations. For example, there are FIFOs which rely on BRAM memory to store many thousands of entries. On the other end of the spectrum, there are PipelineFIFOs which can accept a new value even if they are full if a value is removed from them in the same cycle. This approach allows a focus on the design first, and perform optimization later when everything is working.

It is important to remember that most high-level features of the language are compile time only and do not incur hardware overhead. Accordingly, hardware can be written as flexible and with as many high-level features as possible. Higher order functions and lists are two of the many useful concepts provided by the language which come for free.

The designer should also make sure that things keep on working. Continuous integration techniques are a very valuable tool in hardware design. Otherwise, new bugs might appear with every change and the designer is never sure if the design is in a good state. Apart from functionality testing, regression testing for performance metrics should also be used.

Another important lesson is to embrace the back-of-the-envelope calculation. If it does not work in theory after roughly estimating the performance, it will not magically work in hardware.

11.2 FUTURE WORK

In any of the subjects touched in this thesis, there are many open research questions. TaPaSCo is still under active and frequent development. A very import research direction is the the optimization of the on board interconnects using a network on chip. Currently, many designs on large FPGA experience routing difficulties as the interconnects of the design scale badly with FPGA size.

Another route of TaPaSCo development is the inclusion of on-hardware scheduling for jobs. Right now the user space application is responsible for all scheduling tasks. This limits performance due to the high PCIe latency, especially if jobs rely on data from previous job executions. On hardware scheduling promises increased flexibility and lowered latency. Current research utilizes standardized technologies such as the Heterogeneous Systems Architecture (HSA) [38].

Apart from FPGA itself, this thesis touched a variety of different subjects which all have open research questions. For instance, the stereo vision core introduced in Chapter 6 works well on its own but is currently not integrated into a full image processing pipeline.

The in-network consensus implementation from Chapter 9 showed very promising results. However, many open questions remain in regards to the fault tolerance of the system. Accordingly, the protocol has to be improved to be able to react on problems such as switch failures.

The in-network processing of database pipelines is faced with similar challenges. The current prototype shows promising results but is in a very early state. The accelerator supports only two pipeline types which cannot be changed dynamically. In the future the accelerator should be automatically configured by the query compiler.

In conclusion, the approaches presented in this thesis show great promise and research into these directions should continue. Furthermore, these are not the only domains that benefit from reconfigurable computing. To make FPGA more accessible research into approaches that make them more accessible such as TaPaSCo should be continued.

BIBLIOGRAPHY

[1] *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 2016. ISBN: 978-1-5090-2020-1. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7491900.

[2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "Tensorflow: A system for large-scale machine learning." In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 265–283.

[3] H. Akinaga and H. Shima. "Resistive Random Access Memory (ReRAM) Based on Metal Oxides." In: *Proc. IEEE* 98.12 (Dec. 2010), pp. 2237–2251.

[4] AMD. *2nd Gen AMD EPYC™ Processors Set New Standard for the Modern Datacenter with Record-Breaking Performance and Significant TCO Savings*. July 2019. URL: https://www.amd.com/en/press-releases/2019-08-07-2nd-gen-amd-epyc-processors-set-new-standard-for-the-modern-datacenter.

[5] Apple. *Mac Pro with Apple Afterburner*. Oct. 2019. URL: https://www.apple.com/mac-pro/.

[6] O. J. Arndt, D. Becker, C. Banz, and H. Blume. "Parallel implementation of real-time semi-global matching on embedded multi-core architectures." In: *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*. July 2013, pp. 56–63. DOI: 10.1109/SAMOS.2013.6621106.

[7] Peter J. Ashenden. *The Designer's Guide to VHDL*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN: 1558606742.

[8] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. "Sharing Memory Robustly in Message-passing Systems." In: *J. ACM* 42.1 (Jan. 1995), pp. 124–142. ISSN: 0004-5411.

[9] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. "C?aSH: Structural Descriptions of Synchronous Hardware Using Haskell." In: *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. Sept. 2010, pp. 714–721. DOI: 10.1109/DSD.2010.21.

[10] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. "Chisel: Constructing hardware in a Scala embedded language." In: *DAC Design Automation Conference 2012*. June 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[11] Stevo Bailey, Adam Izraelevitz, Richard Lin, Chick Markley, Paul Rigge, and Edward Wang. *Chisel Bootcamp*. July 2017. URL: https://github.com/freechipsproject/chisel-bootcamp.

[12] C. Banz, S. Hesselbarth, H. Flatt, H. Blume, and P. Pirsch. "Real-time stereo vision system using semi-global matching disparity estimation: Architecture and FPGA-implementation." In: *Embedded Computer Systems (SAMOS), 2010 International Conference on*. July 2010, pp. 93–101. DOI: 10.1109/ICSAMOS.2010.5642077.

[13] Fabrice Bellard. *FFmpeg*. Oct. 2019. URL: http://ffmpeg.org.

[14] BlanchardJ. *FIR Filter.svg*. Wikipedia. 2008. URL: https://commons.wikimedia.org/wiki/File:FIR_Filter.svg.

[15] Marcel Blöcher, Tobias Ziegler, Carsten Binnig, and Patrick Eugster. "Boosting scalable data analytics with modern programmable networks." In: *Proceedings of the 14th International Workshop on Data Management on New Hardware, Houston, TX, USA, June 11, 2018*. 2018, 1:1–1:3.

[16] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN." In: *ACM CCR* 43.4 (Aug. 2013), pp. 99–110.

[17] Pat Bosshart et al. "P4: Programming Protocol-independent Packet Processors." In: *ACM SIGCOMM Computer Communication Review* 44.3 (July 2014), pp. 87–95. URL: http://doi.acm.org/10.1145/2656877.2656890.

[18] Valentino Braitenberg and Almut Schüz. *Cortex: statistics and geometry of neuronal connectivity*. Springer Science & Business Media, 2013.

[19] Geoffrey W Burr, Rohit S Shenoy, Kumar Virwani, Pritish Narayanan, Alvaro Padilla, Bülent Kurdi, and Hyunsang Hwang. "Access Devices for 3D Crosspoint Memory." In: *J. Vac. Sci. Technol. B* 32.4 (July 2014). art. ID 040802.

[20] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems." In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '11. Monterey, CA, USA: ACM, 2011, pp. 33–

36. ISBN: 978-1-4503-0554-9. DOI: 10.1145/1950413.1950423. URL: http://doi.acm.org/10.1145/1950413.1950423.

[21] David de la Chevallerie, Jens Korinth, and Andreas Koch. "ffLink: A Lightweight High-Performance Open-Source PCI Express Gen3 Interface for Reconfigurable Accelerators." In: *SIGARCH Comput. Archit. News* 43.4 (Apr. 2016), pp. 34–39. ISSN: 0163-5964. DOI: 10.1145/2927964.2927971. URL: http://doi.acm.org/10.1145/2927964.2927971.

[22] D. Chiou. "The microsoft catapult project." In: *2017 IEEE International Symposium on Workload Characterization (IISWC)*. Oct. 2017, pp. 124–124. DOI: 10.1109/IISWC.2017.8167769.

[23] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs." In: *Acm sigplan notices* 46.4 (2011), pp. 53–64.

[24] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, and Robert W. Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. UK: Strathclyde Academic Media, 2014. ISBN: 099297870X, 9780992978709.

[25] Leonardo Dagum and Ramesh Menon. "OpenMP: An industry-standard API for shared-memory programming." In: *Computing in Science & Engineering* 1 (1998), pp. 46–55.

[26] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. "Paxos Made Switch-y." In: *SIGCOMM Comput. Commun. Rev.* 46.2 (May 2016), pp. 18–24. ISSN: 0146-4833. DOI: 10.1145/2935634.2935638. URL: http://doi.acm.org/10.1145/2935634.2935638.

[27] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. "Paxos Made Switch-y." In: *ACM CCR* 46.2 (May 2016), pp. 18–24.

[28] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. "NetPaxos: Consensus at Network Speed." In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. SOSR '15. ACM, June 2015. DOI: 10.1145/2774993.2774999.

[29] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. "NetPaxos: Consensus at Network Speed." In: *ACM SOSR*. ACM, June 2015, pp. 1–7.

[30] T. Dang, Jaco Hofmann, Y. Liu, M. Radi, D. Vucinic, and F. Pedone. "Consensus for Non-volatile Main Memory." In: *2018 IEEE 26th International Conference on Network Protocols (ICNP)*. 2018.

[31] Jan Decaluwe. "MyHDL: a Python-Based Hardware Description Language." In: *Linux journal* 127 (2004), pp. 84–87.

[32] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. "Implementation Techniques for Main Memory Database Systems." In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. SIGMOD '84. Boston, Massachusetts: ACM, 1984, pp. 1–8. ISBN: 0-89791-128-8. DOI: 10.1145/602259.602261. URL: http://doi.acm.org/10.1145/602259.602261.

[33] Larry Doolittle. *VHD2VL 3.0*. 2018. URL: https://github.com/ldoolitt/vhd2vl.

[34] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. *From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming*. 2011.

[35] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. "Reducing DRAM Footprint with NVM in Facebook." In: *Eurosys*. Apr. 2018, pp. 1–13.

[36] Daniel Firestone et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud." In: *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*. 2018, pp. 51–66.

[37] Jeffrey Fong, Xiang Wang, Yaxuan Qi, Jun Li, and Weirong Jiang. "ParaSplit: A scalable architecture on FPGA for terabit packet classification." In: *20th Annual Symposium on High-Performance Interconnects*. IEEE, Aug. 2012, pp. 1–8.

[38] HSA Foundation. *HSA Foundation - Harmonizing the Industry around Heterogeneous Computing*. 2019. URL: http://www.hsafoundation.com.

[39] Andreas Geiger, Philip Lenz, and Raquel Urtasun. "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite." In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2012.

[40] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J Brown, Arvind K Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. "Hardware system synthesis from domain-specific languages." In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2014, pp. 1–8.

[41] Wulfram Gerstner and Werner M Kistler. *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press, 2002.

[42] M. B. Gokhale and J. M. Stone. "NAPA C: compiling for a hybrid RISC/FPGA architecture." In: *Proceedings. IEEE Symposium on FPGAs for Custom Computing Machines (Cat. No.98TB100251)*. Apr. 1998, pp. 126–135. DOI: 10.1109/FPGA.1998.707890.

[43] J. Gray. "GRVI Phalanx: A Massively Parallel RISC-V FPGA Accelerator Accelerator." In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. May 2016, pp. 17–20. DOI: 10.1109/FCCM.2016.12.

[44] Sarah Harris and David Harris. *Digital Design and Computer Architecture: ARM Edition*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015. ISBN: 0128000562, 9780128000564.

[45] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. "Darkroom: compiling high-level image processing code into hardware pipelines." In: *ACM Trans. Graph.* 33.4 (2014), pp. 144–1.

[46] Carsten Heinz, Yannick Lavan, Jaco Hofmann, and Andreas Koch. "A Catalog and In-Hardware Evaluation of Open-Source Drop-In Compatible RISC-V Softcore Processors." In: *IEEE Proc. International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE. 2019.

[47] Daniel Hernandez-Juarez, Alejandro Chacón, Antonio Espinosa, David Vázquez, Juan Carlos Moure, and Antonio M López. "Embedded real-time stereo estimation via semi-global matching on the GPU." In: *Procedia Computer Science* 80 (2016), pp. 143–153.

[48] Heiko Hirschmüller. "Accurate and efficient stereo processing by semi-global matching and mutual information." In: *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*. Vol. 2. June 2005, 807–814 vol. 2. DOI: 10.1109/CVPR.2005.56.

[49] Heiko Hirschmüller. "Stereo Processing by Semiglobal Matching and Mutual Information." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30.2 (2008), pp. 328–341. ISSN: 0162-8828. DOI: http://doi.ieeecomputersociety.org/10.1109/TPAMI.2007.1166.

[50] A Hodgkin and A Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve." In: *The Journal of physiology* 117.4 (1952), pp. 500–544.

[51] Jaco Hofmann. "Real-Time Multi-Chip Neural Network for Cognitive Systems." In: ed. by A. Zjajo and R. van Leuken. Real-Time Multi-Chip Neural Network for Cognitive Systems. River Publishers, 2019. Chap. MultiChip Dataflow Architecture for Massive Scale Biophysically Accurate Neuron Simulation.

[52] Jaco Hofmann, Jens Korinth, and Andreas Koch. "A Scalable High-Performance Hardware Architecture for Real-Time Stereo Vision by Semi-Global Matching." In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. Best Paper Runner-Up. 2016.

[53] Jaco Hofmann, Jens Korinth, and Andreas Koch. "A Scalable Latency-Insensitive Architecture for FPGA-Accelerated Semi-Global Matching in Stereo Vision Applications." In: *IEEE Proc. International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2016.

[54] Jaco Hofmann, Lasse Thostrup, Tobias Ziegler, Carsten Binnig, and Andreas Koch. "High-Performance In-Network Data Processing." In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2019, Los Angeles, United States.* 2019.

[55] Jaco Hofmann, A. Zjajo, and R. van Leuken. "Multi-chip dataflow architecture for massive scale biophysically accurate neuron simulation." In: *38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. 2016.

[56] D. Honegger, H. Oleynikova, and M. Pollefeys. "Real-time and low latency embedded computer vision hardware based on a combination of FPGA and mobile CPU." In: *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. Sept. 2014, pp. 4930–4935. DOI: 10.1109/IROS.2014.6943263.

[57] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch. "Hardware/software co-compilation with the Nymble system." In: *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. July 2013, pp. 1–8. DOI: 10.1109/ReCoSoC.2013.6581538.

[58] Google Inc. *Protocol Buffers*. Oct. 2019. URL: https://developers.google.com/protocol-buffers/.

[59] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. "Consensus in a Box: Inexpensive Coordination in Hardware." In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, Mar. 2016, pp. 425–438. ISBN: 978-1-931971-29-4. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/istvan.

[60] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. "Consensus in a Box: Inexpensive Coordination in Hardware." In: *USENIX NSDI*. Mar. 2016, pp. 425–438.

[61] Eugene M Izhikevich. "Which model to use for cortical spiking neurons?" In: *IEEE transactions on neural networks* 15.5 (2004), pp. 1063–1070.

[62] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. "NetChain: Scale-Free Sub-RTT Coordination." In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 35–49. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/jin.

[63] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. "NetChain: Scale-Free Sub-RTT Coordination." In: *USENIX NSDI*. Apr. 2018, pp. 35–49.

[64] Xin Jin, Xiaozhou li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. "NetCache: Balancing Key-Value Stores with Fast In-Network Caching." In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. ACM, Oct. 2017, pp. 121–136. DOI: 10.1145/3132747.3132764.

[65] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. "Compiling Packet Programs to Reconfigurable Switches." In: *USENIX NSDI*. May 2015, pp. 103–115.

[66] J. A. Katine, F. J. Albert, R. A. Buhrman, E. B. Myers, and D. C. Ralph. "Current-Driven Magnetization Reversal and Spin-Wave Excitations in Co /Cu /Co Pillars." In: *Phys. Rev. Lett.* 84 (14 Apr. 2000), pp. 3149–3152.

[67] DerChang Kau, Stephen Tang, Ilya V Karpov, Rick Dodge, Brett Klehn, Johannes A Kalb, Jonathan Strand, Aleshandre Diaz, Nelson Leung, Jack Wu, et al. "A Stackable Cross Point Phase Change Memory." In: *IEEE IEDM*. Dec. 2009, pp. 1–4.

[68] J. Korinth, D. d. l. Chevallerie, and A. Koch. "An Open-Source Tool Flow for the Composition of Reconfigurable Hardware Thread Pool Architectures." In: *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. May 2015, pp. 195–198. DOI: 10.1109/FCCM.2015.22.

[69] Jens Korinth, Jaco Hofmann, Carsten Heinz, and Andreas Koch. "The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems." In: *International Symposium on Applied Reconfigurable Computing (ARC)*. 2019.

[70] J Kruger and F Aiple. "Multimicroelectrode investigation of monkey striate cortex: spike train correlations in the infragranular layers." In: *Journal of Neurophysiology* 60.2 (1988), pp. 798–828.

[71] Ian Kuon, Russell Tessier, Jonathan Rose, et al. "FPGA architecture: Survey and challenges." In: *Foundations and Trends® in Electronic Design Automation* 2.2 (2008), pp. 135–253.

[72] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563.

[73] Leslie Lamport. "The Part-time Parliament." In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

[74] Leslie Lamport. "The Part-time Parliament." In: *ACM TOCS* 16.2 (May 1998), pp. 133–169.

[75] C. Lavin and A. Kaviani. "RapidWright: Enabling Custom Crafted Implementations for FPGAs." In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 2018, pp. 133–140. DOI: 10.1109/FCCM.2018.00030.

[76] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. "RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs." In: *2011 21st International Conference on Field Programmable Logic and Applications*. Sept. 2011, pp. 349–355. DOI: 10.1109/FPL.2011.69.

[77] Alberto Lerner, Rana Hussein, and Philippe Cudré-Mauroux. "The Case for Network Accelerated Query Processing." In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. 2019.

[78] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. "Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering." In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 467–483. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/li.

[79] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. "Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering." In: *USENIX OSDI*. Nov. 2016, pp. 467–483.

[80] D Loke, TH Lee, WJ Wang, LP Shi, R Zhao, YC Yeo, TC Chong, and SR Elliott. "Breaking The Speed Limits of Phase-Change Memory." In: *Science* 336.6088 (Nov. 2012), pp. 1566–1569.

[81] Clive Maxfield. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. 1st. Newton, MA, USA: Newnes, 2004. ISBN: 0750676043, 9780750676045.

[82] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: Enabling Innovation in Campus Networks." In: *ACM SIGCOMM Computer Communication Review* 38.2 (Apr. 2008), pp. 69–74. DOI: 10.1145/1355734.1355746.

[83] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. "SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs." In: *Proceedings of the 2017 ACM SIGCOMM Conference*. SIGCOMM '17. ACM, Aug. 2017, pp. 15–28. DOI: 10.1145/3098822.3098824.

[84] M. Michael, J. Salmen, J. Stallkamp, and M. Schlipsing. "Real-time stereo vision: Optimizing Semi-Global Matching." In: *Intelligent Vehicles Symposium (IV), 2013 IEEE*. June 2013, pp. 1197–1202. DOI: 10.1109/IVS.2013.6629629.

[85] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System." In: *ACM/IEEE HPC*. Nov. 2010, pp. 1–11.

[86] Sascha Mühlbach, M. Brunner, C. Roblee, and Andreas Koch. "MalCoBox: Designing a 10 Gb/s Malware Collection Honeypot Using Reconfigurable Technology." In: *IEEE Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*. 2010.

[87] Sascha Muhlbach and Andreas Koch. "A dynamically reconfigured network platform for high-speed malware collection." In: *2010 International Conference on Reconfigurable Computing and FPGAs*. IEEE. 2010, pp. 79–84.

[88] M. Naylor and S. Moore. "A generic synthesisable test bench." In: *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*. Sept. 2015, pp. 128–137. DOI: 10.1109/MEMCOD.2015.7340479.

[89] BAREFOOT NETWORKS. *Tofino Programmable Switch*. URL: https://www.barefootnetworks.com/technology/.

[90] R. Nikhil. "Bluespec System Verilog: efficient, correct RTL from high level specifications." In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* June 2004, pp. 69–70. DOI: 10.1109/MEMCOD.2004.1459818.

[91] E. Nurvitadhi, Jaewoong Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr. "Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC." In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016, pp. 1–4. DOI: 10.1109/FPL.2016.7577314.

[92] J. Oppermann, A. Koch, T. Yu, and O. Sinnen. "Domain-specific optimisation for the high-level synthesis of CellML-based simulation accelerators." In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2015, pp. 1–7. DOI: 10.1109/FPL.2015.7294019.

[93] Stanford R Ovshinsky. "Reversible Electrical Switching Phenomena in Disordered Structures." In: *Phys. Rev. Lett.* 21.20 (Nov. 1968), p. 1450.

[94] Samir Palnitkar. *Verilog®Hdl: A Guide to Digital Design and Synthesis, Second Edition*. Second. Upper Saddle River, NJ, USA: Prentice Hall Press, 2003. ISBN: 0-13-044911-3.

[95] Charles Papon. *SpinalHDL*. Github. 2014. URL: https://github.com/SpinalHDL/SpinalHDL.

[96] Fabio Pellizzer and Agostino Pirovano. "Phase Change Memory with Ovonic Threshold Switch." 7677830. Mar. 2010.

[97] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. "Designing Distributed Systems Using Approximate Synchrony in Data Center Networks." In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 43–57. ISBN: 978-1-931971-218. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ports.

[98] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. "Designing Distributed Systems Using Approximate Synchrony in Data Center Networks." In: *USENIX NSDI*. May 2015, pp. 43–57.

[99] A. Qamar, C. Passerone, L. Lavagno, and F. Gregoretti. "Design space exploration of a stereo vision system using high-level synthesis." In: *Mediterranean Electrotechnical Conference (MELECON), 2014 17th IEEE*. Apr. 2014, pp. 500–504. DOI: 10.1109/MELCON.2014.6820585.

[100] Robbert van Renesse and Fred B. Schneider. "Chain Replication for Supporting High Throughput and Availability." In: *USENIX OSDI*. Dec. 2004, pp. 7–7.

[101] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. "High-Speed Query Processing over High-Speed Networks." In: *PVLDB* 9.4 (2015), pp. 228–239.

[102] F. M. Sánchez, R. Mateos, E. J. Bueno, J. Mingo, and I. Sanz. "Comparative of HLS and HDL implementations of a grid synchronization algorithm." In: *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*. Nov. 2013, pp. 2232–2237. DOI: 10.1109/IECON.2013.6699478.

[103] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. "In-Network Computation is a Dumb Idea Whose Time Has Come." In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017*. 2017, pp. 150–156.

[104] D. Scharstein and R. Szeliski. "High-accuracy stereo depth maps using structured light." In: *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*. Vol. 1. June 2003, pp. 195–202. DOI: 10.1109/CVPR.2003.1211354.

[105] Daniel Scharstein. *Middlebury Stereo Evaluation - Version 3*. 2019. URL: http://vision.middlebury.edu/stereo/eval3/.

[106] Daniel Scharstein and Richard Szeliski. "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms." In: *Int. J. Comput. Vision* 47.1-3 (Apr. 2002), pp. 7–42. ISSN: 0920-5691. DOI: 10.1023/A:1014573219977. URL: http://dx.doi.org/10.1023/A:1014573219977.

[107] K. Schmid and H. Hirschmüller. "Stereo vision and IMU based real-time ego-motion and depth image computation on a handheld device." In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. May 2013, pp. 4671–4678. DOI: 10.1109/ICRA.2013.6631242.

[108] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. "DRAM Errors in the Wild: A Large-scale Field Study." In: *PER* 37.1 (June 2009), pp. 193–204.

[109] Giorgio Servalli. "A 45nm Generation Phase Change Memory Technology." In: *IEEE IEDM*. Dec. 2009, pp. 1–4.

[110] Roy R Shanks. "Ovonic Threshold Switching Characteristics." In: *J. Non-Cryst. Solids* 2 (Jan. 1970), pp. 504–514.

[111] Naveen Kr Sharma, Antoine Kaufmann, Thomas E Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. "Evaluating the Power of Flexible Packet Processing for Network Resource Allocation." In: *USENIX NSDI*. Mar. 2017, pp. 67–82.

[112] Hooman Shayani, Peter J Bentley, and Andy M Tyrrell. "Hardware implementation of a bio-plausible neuron model for evolution and growth of spiking neural networks on FPGA." In: *2008 NASA/ESA Conference on Adaptive Hardware and Systems*. IEEE. 2008, pp. 236–243.

[113] Georgios Smaragdos, Sebastian Isaza, Martijn F van Eijk, Ioannis Sourdis, and Christos Strydis. "FPGA-based biophysically-meaningful modeling of olivocerebellar neurons." In: *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. ACM. 2014, pp. 89–98.

[114] Leonardo Solis-Vasquez and Andreas Koch. "A Case Study in Using OpenCL on FPGAs: Creating an Open-Source Accelerator of the AutoDock Molecular Docking Software." In: *Fifth International Workshop on FPGAs for Software Programmers (FSP)*. 2018.

[115] Lukas Sommer, Julian Oppermann, Alejandro Molina, Carsten Binnig, Kristian Kersting, and Andreas Koch. "Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-based Accelerators." In: *IEEE International Conference on Computer Design (ICCD)*. 2018.

[116] R. Spangenberg, T. Langner, S. Adfeldt, and R. Rojas. "Large scale Semi-Global Matching on the CPU." In: *Intelligent Vehicles Symposium Proceedings, 2014 IEEE*. June 2014, pp. 195–201. DOI: 10.1109/IVS.2014.6856419.

[117] J. E. Stone, D. Gohara, and G. Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems." In: *Computing in Science Engineering* 12.3 (May 2010), pp. 66–73. DOI: 10.1109/MCSE.2010.69.

[118] Chao Sun, Damien Le Moal, Qingbo Wang, Robert Mateescu, Filip Blagojevic, Martin Lueker-Boden, Cyril Guyot, Zvonimir Bandic, and Dejan Vucinic. "Latency Tails of Byte-Addressable Non-Volatile Memories in Systems." In: *IMW*. IEEE. May 2017, pp. 1–4.

[119] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman. "LaKe: The Power of In-Network Computing." In: *2018 International Conference on ReConFigurable Computing and FPGAs*. ReConFil 2018. IEEE, Dec. 2018. DOI: 10.1109/RECONFIG.2018.8641696.

[120] D. Tong, S. Zhou, and V. K. Prasanna. "High-Throughput Online Hash Table on FPGA." In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. May 2015, pp. 105–112. DOI: 10.1109/IPDPSW.2015.149.

[121] Paul Viola and William M. Wells III. "Alignment by Maximization of Mutual Information." In: *Int. J. Comput. Vision* 24.2 (Sept. 1997), pp. 137–154. ISSN: 0920-5691. DOI: 10.1023/A:1007958904918. URL: http://dx.doi.org/10.1023/A:1007958904918.

[122] Wenqiang Wang, Jing Yan, Ningyi Xu, Yu Wang, and Feng-Hsiung Hsu. "Real-time high-quality stereo vision system in FPGA." In: *Field-Programmable Technology (FPT), 2013 International Conference on*. Dec. 2013, pp. 358–361. DOI: 10.1109/FPT.2013.6718387.

[123] Wei Liang, Wenbo Yin, Ping Kang, and Lingli Wang. "Memory efficient and high performance key-value store on FPGA using Cuckoo hashing." In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016, pp. 1–4. DOI: 10.1109/FPL.2016.7577355.

[124] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. "Phase Change Memory." In: *Proc. IEEE* 98.12 (Dec. 2010), pp. 2201–2227.

[125] Matthias Wuttig and Noboru Yamada. "Phase-Change Materials for Rewriteable Data Storage." In: *Nature Mater.* 6.11 (2007), p. 824.

[126] Xilinx. *UltraScale Architecture Configurable Logic Block User Guide (UG574)*. Tech. rep. Feb. 2017. URL: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf.

[127] "Xilinx Announces Vitis - a Unified Software Platform Unlocking a New Design Experience for All Developers." In: *Xilinx Announces Vitis - a Unified Software Platform Unlocking a New Design Experience for All Developers* (Oct. 2019). URL: https://www.xilinx.com/news/press/2019/xilinx-announces-vitis--a-unified-software-platform-unlocking-a-new-design-experience-for-all-developers.html.

[128] "Xilinx XC2064—world's first commercial FPGA—inducted into IEEE's Chip Hall of Fame today." In: *Xilinx XC2064—world's first commercial FPGA—inducted into IEEE's Chip Hall of Fame today* (June 2017). URL: https://forums.xilinx.com/t5/Xcell-Daily-Blog-Archived/Xilinx-XC2064-world-s-first-commercial-FPGA-inducted-into-IEEE-s/ba-p/775806.

[129] *XPliant Ethernet Switch Product Family*. www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html. 2014.

[130] Ramin Zabih and John Woodfill. "Computer Vision — ECCV '94: Third European Conference on Computer Vision Stockholm, Sweden, May 2–6 1994 Proceedings, Volume II." In: ed. by Jan-Olof Eklundh. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994. Chap. Non-parametric local transforms for computing visual correspondence, pp. 151–158. ISBN: 978-3-540-48400-4. DOI: 10.1007/BFb0028345. URL: http://dx.doi.org/10.1007/BFb0028345.

[131] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. "The End of a Myth: Distributed Transaction Can Scale." In: *PVLDB* 10.6 (2017), pp. 685–696. DOI: 10.14778/3055330.3055335. URL: http://www.vldb.org/pvldb/vol10/p685-zamanian.pdf.

[132] Tobias Ziegler, Carsten Binnig, and Uwe Röhm. "Skew-resilient Query Processing for Fast Networks." In: *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Workshopband*. 2019, pp. 81–85.

[133] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. "NetFPGA SUME: Toward 100 Gbps as Research Commodity." In: *IEEE Micro* 34.5 (Sept. 2014), pp. 32–41. ISSN: 0272-1732. DOI: 10.1109/MM.2014.61.

[134] Amir Zjajo, Jaco Hofmann, Gerrit Jan Christiaanse, Martijn Van Eijk, Georgios Smaragdos, Christos Strydis, Alexander de Graaf, Carlo Galuzzi, and Rene van Leuken. "A real-time reconfigurable multichip architecture for large-scale biophysically accurate neuron simulation." In: *IEEE transactions on biomedical circuits and systems* 12.2 (2018), pp. 326–337.

ERKLÄRUNGEN LAUT PROMOTIONSORDNUNG

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, December 2019

Jaco Hofmann